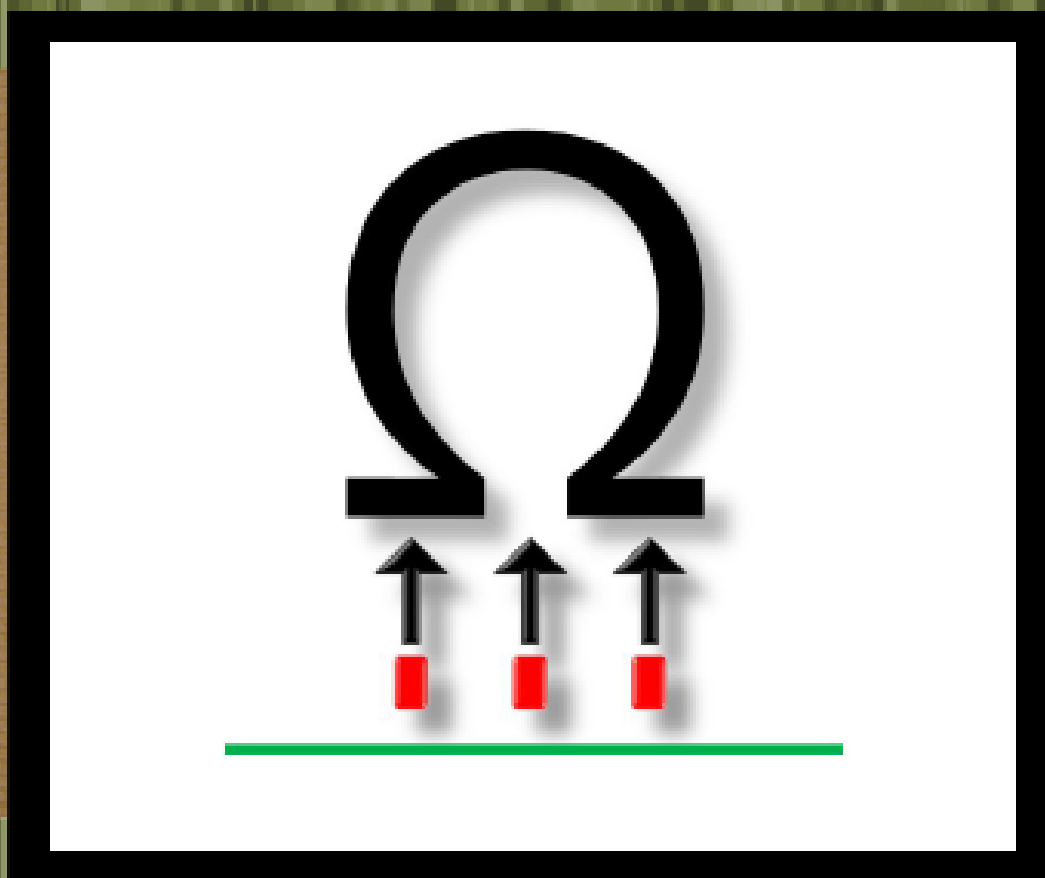# Competitive Programming 3

The New Lower Bound of Programming Contests.

Steven Halim

Felix Halim

HANDBOOK FOR ACM ICPC AND IOI CONTESTANTS
2013

Contents

Foreword

A long time ago (on the 11th of November in 2003, Tuesday, 3:55:57 UTC), I received an e-mail with the following message:

> "I should say in a simple word that with the UVa Site, you have given birth to a new CIVILIZATION and with the books you write (he meant "Programming Challenges: The Programming Contest Training Manual" [60], coauthored with Steven Skiena), you inspire the soldiers to carry on marching. May you live long to serve the humanity by producing super-human programmers."

Although that was clearly an exaggeration, it did cause me to think. I had a dream: to create a community around the project I had started as a part of my teaching job at UVa, with people from all around the world working together towards the same ideal. With a little searching, I quickly found a whole online community running a web-ring of sites with excellent tools that cover and provide whatever the UVa site lacked.

To me, 'Methods to Solve' by Steven Halim, a very young student from Indonesia, was one of the more impressive websites. I was inspired to believe that the dream would become real one day, because in this website lay the result of the hard work of a genius of algorithms and informatics. Moreover, his declared objectives matched the core of my dream: to serve humanity. Even better, he has a brother with similar interests and capabilities, Felix Halim.

It's a pity that it takes so much time to start a real collaboration, but life is like that. Fortunately, all of us have continued working together in a parallel fashion towards the realization of that dream—the book that you have in your hands now is proof of that.

I can't imagine a better complement for the UVa Online Judge. This book uses lots of examples from UVa carefully selected and categorized both by problem type and solving technique, providing incredibly useful help for the users of the site. By mastering and practicing most programming exercises in this book, a reader can easily solve at least 500 problems in the UVa Online Judge, which will place them in the top 400-500 amongst ≈100000 UVa OJ users.

It's clear that the book "Competitive Programming: Increasing the Lower Bound of Programming Contests" is suitable for programmers who want to improve their ranks in upcoming ICPC regionals and IOIs. The two authors have gone through these contests (ICPC and IOI) themselves as contestants and now as coaches. But it's also an essential colleague for newcomers—as Steven and Felix say in the introduction 'the book is not meant to be read once, but several times'.

Moreover, it contains practical C++ source code to implement given algorithms. Understanding a problem is one thing, but knowing the algorithm to solve it is another, and implementing the solution well in short and efficient code is tricky. After you have read this extraordinary book three times you will realize that you are a much better programmer and, more importantly, a happier person.

@ ACM ICPC World Finals Warsaw 2012

L-R: Fredrik Niemelä, Carlos, Miguel Revilla, Miguel Jr, Felix, Steven

Miguel A. Revilla, University of Valladolid
UVa Online Judge site creator;
ACM-ICPC International Steering Committee Member and Problem Archivist
http://uva.onlinejudge.org; http://livearchive.onlinejudge.org

Preface

This book is a must have for every competitive programmer. Mastering the contents of this book is a necessary (but maybe not sufficient) condition if one wishes to take a leap forward from being just another ordinary coder to being among one of the world's finest programmers.

Typical readers of this book would include:

1. University students who are competing in the annual ACM International Collegiate Programming Contest (ICPC) [66] Regional Contests (including the World Finals),

2. Secondary or High School Students who are competing in the annual International Olympiad in Informatics (IOI) [34] (including the National or Provincial Olympiads),

3. Coaches who are looking for comprehensive training materials for their students [24],

4. Anyone who loves solving problems through computer programs. There are numerous programming contests for those who are no longer eligible for ICPC, including TopCoder Open, Google CodeJam, Internet Problem Solving Contest (IPSC), etc.

Prerequisites

This book is *not* written for novice programmers. This book is aimed at readers who have at least basic knowledge in programming methodology, are familiar with at least one of these programming languages (C/C++ or Java, preferably both), have passed a basic data structures and algorithms course (typically taught in year one of Computer Science university curricula), and understand simple algorithmic analysis (at least the big-O notation). In the third edition, more content has been added so that this book can also be used as a *supplementary reading* for a basic *Data Structures and Algorithms* course.

To ACM ICPC Contestants

Zi Chun Koh, Trinh Tuan Phuong, Harta Wijaya
Steven Halim

We know that one cannot probably win the ACM ICPC regional just by mastering the contents of the *current version (third edition)* of this book. While we have included a lot of materials in this book—much more than in the first two editions—we are aware that much more than what this book can offer is required to achieve that feat. Some additional pointers to useful references are listed in the chapter notes for readers who are hungry for more. We believe, however, that your team will fare much better in future ICPCs after mastering the contents of this book. We hope that this book will serve as both inspiration and motivation for your 3-4 year journey competing in ACM ICPCs during your University days.

To IOI Contestants

L-R: Daniel, Mr Cheong, Raymond, Steven, Zhan Xiong, Dr Roland, Chuanqi

Much of our advice for ACM ICPC contestants applies to you too. The ACM ICPC and IOI syllabi are largely similar, except that IOI, *for now*, currently excludes the topics listed in the following Table 1. You can skip these items until your university years (when you join that university's ACM ICPC teams). However, learning these techniques in advance may definitely be beneficial as some tasks in IOI can become easier with additional knowledge.

We know that one cannot win a medal in IOI just by mastering the contents of the *current version (third edition)* of this book. While we believe that many parts of the IOI syllabus has been included in this book—hopefully enabling you to achieve a respectable score in future IOIs—we are well aware that modern IOI tasks require keen problem solving skills and tremendous creativity—virtues that we cannot possibly impart through this static textbook. This book can provide knowledge, but the hard work must ultimately be done by you. With practice comes experience, and with experience comes skill. So, keep practicing!

Topic	In This Book
Data Structures: Union-Find Disjoint Sets	Section 2.4.2
Graph: Finding SCCs, Network Flow, Bipartite Graphs	Section 4.2.1, 4.6.3, 4.7.4
Math: BigInteger, Probability Theory, Nim Games	Section 5.3, 5.6, 5.8
String Processing: Suffix Trees/Arrays	Section 6.6
More Advanced Topics: A*/IDA*	Section 8.2
Many of the Rare Topics	Chapter 9

Table 1: Not in IOI Syllabus [20] *Yet*

To Teachers and Coaches

This book is used in Steven's CS3233 - 'Competitive Programming' course in the School of Computing at the National University of Singapore. CS3233 is conducted in 13 teaching weeks using the following lesson plan (see Table 2). The PDF slides (only the public version) are given in the companion web site of this book. Fellow teachers/coaches should feel free to modify the lesson plan to suit students' needs. Hints or brief solutions of the **non-starred** written exercises in this book are given at the back of each chapter. Some of the **starred** written exercises are quite challenging and have neither hints nor solutions. These can probably be used as exam questions or contest problems (of course, solve them first!).

This book is also used as a supplementary reading in Steven's CS2010 - 'Data Structures and Algorithms' course, mostly for the implementation of several algorithms and written/programming exercises.

Wk	Topic	In This Book
01	Introduction	Ch 1, Sec 2.2, 5.2, 6.2-6.3, 7.2
02	Data Structures & Libraries	Chapter 2
03	Complete Search, Divide & Conquer, Greedy	Section 3.2-3.4; 8.2
04	Dynamic Programming 1 (Basic ideas)	Section 3.5; 4.7.1
05	Dynamic Programming 2 (More techniques)	Section 5.4; 5.6; 6.5; 8.3
06	Mid-Semester Team Contest	Chapter 1 - 4; parts of Ch 9
-	Mid-Semester Break	(homework)
07	Graph 1 (Network Flow)	Section 4.6; parts of Ch 9
08	Graph 2 (Matching)	Section 4.7.4; parts of Ch 9
09	Mathematics (Overview)	Chapter 5
10	String Processing (Basic skills, Suffix Array)	Chapter 6
11	(Computational) Geometry (Libraries)	Chapter 7
12	More Advanced Topics	Section 8.4; parts of Ch 9
13	Final Team Contest	Chapter 1-9 and maybe more
-	No final exam	-

Table 2: Lesson Plan of Steven's CS3233

For *Data Structures and Algorithms* Courses

The contents of this book have been expanded in this edition so that the *first four* chapters of this book are more accessible to *first year* Computer Science students. Topics and exercises that we have found to be relatively difficult and thus unnecessarily discouraging for first timers have been moved to the now bulkier Chapter 8 or to the new Chapter 9. This way, students who are new to Computer Science will perhaps not feel overly intimidated when they peruse the first four chapters.

Chapter 2 has received a major update. Previously, Section 2.2 was just a casual list of classical data structures and their libraries. This time, we have expanded the write-up and added lots of written exercises so that this book can also be used to support a *Data Structures* course, especially in the terms of *implementation* details.

The four problem solving paradigms discussed in Chapter 3 appear frequently in typical *Algorithms* courses. The text in this chapter has been expanded and edited to help new Computer Science students.

Parts of Chapter 4 can also be used as a supplementary reading or *implementation* guide to enhance a *Discrete Mathematics* [57, 15] or a basic *Algorithms* course. We have also provide some new insights on viewing Dynamic Programming techniques as algorithms on DAGs. Such discussion is currently still regrettably uncommon in many Computer Science textbooks.

To All Readers

Due to its diversity of coverage and depth of discussion, this book is *not* meant to be read once, but several times. There are many written (≈ 238) and programming exercises (≈ 1675) listed and spread across almost every section. You can skip these exercises at first if the solution is too difficult or requires further knowledge and technique, and revisit them after studying other chapters of this book. Solving these exercises will strengthen your understanding of the concepts taught in this book as they usually involve interesting applications, twists or variants of the topic being discussed. Make an effort to attempt them—time spent solving these problems will definitely not be wasted.

We believe that this book is and will be relevant to many university and high school students. Programming competitions such as the ICPC and IOI are here to stay, at least for many years ahead. New students should aim to understand and internalize the basic knowledge presented in this book before hunting for further challenges. However, the term 'basic' might be slightly misleading—please check the table of contents to understand what we mean by 'basic'.

As the title of this book may imply, the purpose of this book is clear: We aim to improve everyone's programming abilities and thus increase the *lower bound* of programming competitions like the ICPC and IOI in the future. With more contestants mastering the contents of this book, we hope that the year 2010 (when the first edition of this book was published) will be a watershed marking an accelerated improvement in the standards of programming contests. We hope to help more teams solve more (≥ 2) problems in future ICPCs and help more contestants to achieve greater (≥ 200) scores in future IOIs. We also hope to see many ICPC and IOI coaches around the world (especially in South East Asia) adopt this book for the aid it provides in mastering topics that students cannot do without in competitive programming contests. If such a proliferation of the required 'lower-bound' knowledge for competitive programming is achieved, then this book's primary objective of advancing the level of human knowledge will have been fulfilled, and we, as the authors of this book, will be very happy indeed.

Convention

There are lots of C/C++ code and also some Java code (especially in Section 5.3) included in this book. If they appear, they will be typeset in `this monospace font`.

For the C/C++ code in this book, we have adopted the frequent use of `typedef`s and macros—features that are commonly used by competitive programmers for convenience, brevity, and coding speed. However, we cannot use similar techniques for Java as it does not contain similar or analogous features. Here are some examples of our C/C++ code shortcuts:

```
// Suppress some compilation warning messages (only for VC++ users)
#define _CRT_SECURE_NO_DEPRECATE
```

```
// Shortcuts for "common" data types in contests
typedef long long      ll;       // comments that are mixed in with code
typedef pair<int, int> ii;         // are aligned to the right like this
typedef vector<ii>     vii;
typedef vector<int>    vi;
#define INF 1000000000    // 1 billion, safer than 2B for Floyd Warshall's


// Common memset settings
//memset(memo, -1, sizeof memo); // initialize DP memoization table with -1
//memset(arr, 0, sizeof arr);                    // to clear array of integers


// We have abandoned the use of "REP" and "TRvii" since the second edition
// in order to reduce the confusion encountered by new programmers
```

The following shortcuts are frequently used in both our C/C++ and Java code:

```
// ans = a ? b : c;            // to simplify: if (a) ans = b; else ans = c;
// ans += val;            // to simplify: ans = ans + val; and its variants
// index = (index + 1) % n;          // index++; if (index >= n) index = 0;
// index = (index + n - 1) % n;   // index--; if (index < 0) index = n - 1;
// int ans = (int)((double)d + 0.5);    // for rounding to nearest integer
// ans = min(ans, new_computation);                  // min/max shortcut
// alternative form but not used in this book: ans <?= new_computation;
// some code use short circuit && (AND) and || (OR)
```

Problem Categorization

As of 24 May 2013, Steven and Felix—combined—have solved 1903 UVa problems ($\approx 46.45\%$ of the entire UVa problemset). About ≈ 1675 of them are discussed and categorized in this book. Since late 2011, some Live Archive problems have also been integrated in the UVa Online Judge. In this book, we use *both* problem numberings, but the primary sort key used in the index section of this book is the UVa problem number.

These problems are categorized according to a *'load balancing'* scheme: If a problem can be classified into two or more categories, it will be placed in the category with a lower number of problems. This way, you may find that some problems have been 'wrongly' categorized, where the category that it appears in might not match the technique that you have used to solve it. We can only guarantee that if you see problem X in category Y, then you know that *we* have managed to solve problem X with the technique mentioned in the section that discusses category Y.

We have also limited each category to at most 25 (TWENTY FIVE) problems, splitting them into separate categories when needed.

If you need hints for any of the problems (that we have solved), flip to the handy index at the back of this book instead of flipping through each chapter—it might save you some time. The index contains a list of UVa/LA problems, ordered by their problem number (do a binary search!) and augmented by the pages that contain discussion of said problems (and the data structures and/or algorithms required to solve that problem). In the third edition, we allow the hints to span more than one line so that they can be more meaningful.

Utilize this categorization feature for your training! Solving at least a few problems from each category (especially the ones we have highlighted as **must try ***) is a great way to diversify your problem solving skillset. For conciseness, we have limited ourselves to a maximum of 3 highlights per category.

Changes for the Second Edition

There are *substantial* changes between the first and the second edition of this book. As the authors, we have learned a number of new things and solved hundreds of programming problems during the one year gap between these two editions. We also have received feedback from readers, especially from Steven's CS3233 class Sem 2 AY2010/2011 students, and have incorporated these suggestions in the second edition.

Here is a summary of the important changes for the second edition:

- The first noticeable change is the layout. We now have a greater information density on each page. The 2nd edition uses single line spacing instead of the 1.5 line spacing used in the 1st edition. The positioning of small figures is also enhanced so that we have a more compact layout. This is to avoid increasing the number of pages by too much while still adding more content.

- Some minor bugs in our code examples (both the ones displayed in the book and the soft copies provided in the companion web site) have been fixed. All code samples now have much more meaningful comments to aid in comprehension.

- Several language-related issues (typographical, grammatical or stylistic) have been corrected.

- Besides enhancing the discussion of many data structures, algorithms, and programming problems, we have also added these *new* materials in each chapter:

  1. Many new Ad Hoc problems to kick start this book (Section 1.4).

  2. A lightweight set of Boolean (bit-manipulation) techniques (Section 2.2), Implicit Graphs (Section 2.4.1), and Fenwick Tree data structures (Section 2.4.4).

  3. More DP: A clearer explanation of bottom-up DP, the $O(n \log k)$ solution for the LIS problem, the 0-1 Knapsack/Subset Sum, and DP TSP (using the bitmask technique) (Section 3.5.2).

  4. A reorganization of the graph material into: Graph Traversal (both DFS and BFS), Minimum Spanning Tree, Shortest Paths (Single-Source and All-Pairs), Maximum Flow, and Special Graphs. New topics include Prim's MST algorithm, a discussion of DP as a traversal on implicit DAGs (Section 4.7.1), Eulerian Graphs (Section 4.7.3), and the Augmenting Path algorithm (Section 4.7.4).

  5. A reorganization of mathematical techniques (Chapter 5) into: Ad Hoc, Java BigInteger, Combinatorics, Number Theory, Probability Theory, Cycle-Finding, Game Theory (new), and Powers of a (Square) Matrix (new). Each topic has been rewritten for clarity.

  6. Basic string processing skills (Section 6.2), more string-related problems (Section 6.3), including string matching (Section 6.4), and an enhanced Suffix Tree/Array explanation (Section 6.6).

  7. More geometry libraries (Chapter 7), especially on points, lines and polygons.

  8. A new Chapter 8, which contains discussion on problem decomposition, advanced search techniques (A*, Depth Limited Search, Iterative Deepening, IDA*), advanced DP techniques (more bitmask techniques, the Chinese Postman Problem, a compilation of common DP states, a discussion of better DP states, and some harder DP problems).

- Many existing figures in this book have been redrawn and enhanced. Many new figures have been added to help explain the concepts more clearly.

- The first edition is mainly written using from the viewpoint of the ICPC contestant and C++ programmer. The second edition is written to be more balanced and includes the IOI perspective. Java support is also strongly enhanced in the second edition. However, we do not support any other programming languages as of yet.

- Steven's 'Methods to Solve' website has now been fully integrated in this book in the form of 'one liner hints' for each problem and the useful problem index at the back of this book. Now, reaching 1000 problems solved in UVa online judge is no longer a wild dream (we believe that this feat is doable by a *serious* 4-year CS university undergraduate).

- Some examples in the first edition use old programming problems. In the second edition, these examples have been replaced/added with newer examples.

- ≈ 600 more programming exercises from the UVa Online Judge and Live Archive have been solved by Steven & Felix and added to this book. We have also added many more written exercises throughout the book with hints/short solutions as appendices.

- Short profiles of data structure/algorithm inventors have been adapted from Wikipedia [71] or other sources for this book. It is nice to know a little bit more about these inventors.

Changes for the Third Edition

We gave ourselves two years (skipping 2012) to prepare a *substantial* number of improvements and additional materials for the third edition of this book. Here is the summary of the important changes for the third edition:

- The third edition now uses a slightly larger font size (12 pt) compared to second edition (11 pt), a 9 percent increase. Hopefully many readers will find the text more readable this time. We also use larger figures. These decisions, however, have increased the number of pages and rendered the book thicker. We have also adjusted the left/right margin in odd/even pages to increase readability.

- The layout has been changed to start almost every section on a new page. This is to make the layout far easier to manage.

- We have added *many more* written exercises throughout the book and classifed them into **non-starred** (for self-checking purposes; hints/solutions are at the back of each chapter) and **starred *** versions (for extra challenges; no solution is provided). The written exercises have been placed close to the relevant discussion in the body text.

- ≈ 477 more programming exercises from the UVa Online Judge and Live Archive have been solved by Steven & Felix and consequently added to this book. We thus have maintained a sizeable ≈ 50% (to be precise, ≈ 46.45%) coverage of UVa Online Judge problems even as the judge has grown in the same period of time. These newer problems have been listed in an *italic font*. Some of the newer problems have replaced older ones as the **must try** problems. All programming exercises are now always placed at the end of a section.

- We now have proof that *capable* CS students can achieve ≥ 500 AC problems (from 0) in the UVa Online Judge in just one University semester (4 months) with this book.

- The *new* (or revised) materials, chapter by chapter:

  1. Chapter 1 contains a gentler introduction for readers who are new to competitive programming. We have elaborated on stricter Input/Output (I/O) formats in typical programming problems and common routines for dealing with them.

  2. We add one more linear data structure: 'deque' in Section 2.2. Chapter 2 now contains a more detailed discussion of almost all data structures discussed in this chapter, especially Section 2.3 and 2.4.

  3. In Chapter 3, we have a more detailed discussions of various Complete Search techniques: Nested loops, generating subsets/permutations iteratively, and recursive backtracking. New: An interesting trick to write and print Top-Down DP solutions, Discussion of Kadane's algorithm for Max 1D Range Sum.

  4. In Chapter 4, we have revised white/gray/black labels (legacy from [7]) to their standard nomenclature, renaming 'max flow' to 'network flow' in the process. We have also referred to the algorithm author's actual scientific paper for a better understanding of the original ideas of the algorithm. We now have new diagrams of the implicit DAG in classical DP problems found in Section 3.5.

  5. Chapter 5: We have included greater coverage of Ad Hoc mathematics problems, a discussion of an interesting Java BigInteger operation: `isProbablePrime`, added/expanded several commonly used Combinatorics formulae and modified sieve algorithms, expanded/revised sections on Probability Theory (Section 5.6), Cycle-finding (Section 5.7), and Game Theory (Section 5.8).

  6. Chapter 6: We rewrite Section 6.6 to have a better explanation of Suffix Trie/Tree/Array by reintroducing the concept of terminating character.

  7. Chapter 7: We trim this chapter into two core sections and improve the library code quality.

  8. Chapter 8: The harder topics that were listed in Chapter 1-7 in the 2nd edition have now been relocated to Chapter 8 (or Chapter 9 below). New: Discussion of harder backtracking routine, State-Space search, meet in the middle, trick of using balanced BST as memo table, and a more comprehensive section about problem decomposition.

  9. New Chapter 9: Various rare topics that appear once a while in programming contests have been added. Some of them are easy, but many of them are hard and can be somewhat important score determinants in programming contests.

Supporting Websites

This book has an official companion web site at `sites.google.com/site/stevenhalim`, from which you can obtain a soft copy of sample source code and the (*public/simpler version) of the*) PDF slides used in Steven's CS3233 classes.

All programming exercises in this book are integrated in the `uhunt.felix-halim.net` tool and can be found in the UVa Online Judge at `uva.onlinejudge.org`

New in the third edition: Many algorithms now have interactive visualizations at:
`www.comp.nus.edu.sg/~stevenha/visualization`

Acknowledgments for the First Edition

From Steven: **I want to thank**

- God, Jesus Christ, and the Holy Spirit, for giving me talent and passion in competitive programming.

- my lovely wife, Grace Suryani, for allowing me to spend our precious time for this project.

- my younger brother and co-author, Felix Halim, for sharing many data structures, algorithms, and programming tricks to improve the writing of this book.

- my father Lin Tjie Fong and mother Tan Hoey Lan for raising us and encouraging us to do well in our study and work.

- the School of Computing, National University of Singapore, for employing me and allowing me to teach the CS3233 - 'Competitive Programming' module from which this book was born.

- NUS/ex-NUS professors/lecturers who have shaped my competitive programming and coaching skills: Prof Andrew Lim Leong Chye, Assoc Prof Tan Sun Teck, Aaron Tan Tuck Choy, Assoc Prof Sung Wing Kin, Ken, Dr Alan Cheng Holun.

- my friend Ilham Winata Kurnia for proof reading the manuscript of the first edition.

- fellow Teaching Assistants of CS3233 and ACM ICPC Trainers @ NUS: Su Zhan, Ngo Minh Duc, Melvin Zhang Zhiyong, Bramandia Ramadhana.

- my CS3233 students in Sem2 AY2008/2009 who inspired me to come up with the lecture notes and students in Sem2 AY2009/2010 who verified the content of the first edition of this book and gave the initial Live Archive contribution

Acknowledgments for the Second Edition

From Steven: **Additionally, I also want to thank**

- the first $\approx$ 550 buyers of the 1st edition as of 1 August 2011 (this number is no longer updated). Your supportive responses encourage us!

- a fellow Teaching Assistant of CS3233 @ NUS: Victor Loh Bo Huai.

- my CS3233 students in Sem2 AY2010/2011 who contributed in both technical and presentation aspects of the second edition, in alphabetical order: Aldrian Obaja Muis, Bach Ngoc Thanh Cong, Chen Juncheng, Devendra Goyal, Fikril Bahri, Hassan Ali Askari, Harta Wijaya, Hong Dai Thanh, Koh Zi Chun, Lee Ying Cong, Peter Phandi, Raymond Hendy Susanto, Sim Wenlong Russell, Tan Hiang Tat, Tran Cong Hoang, Yuan Yuan, and one other student who prefers to be anonymous.
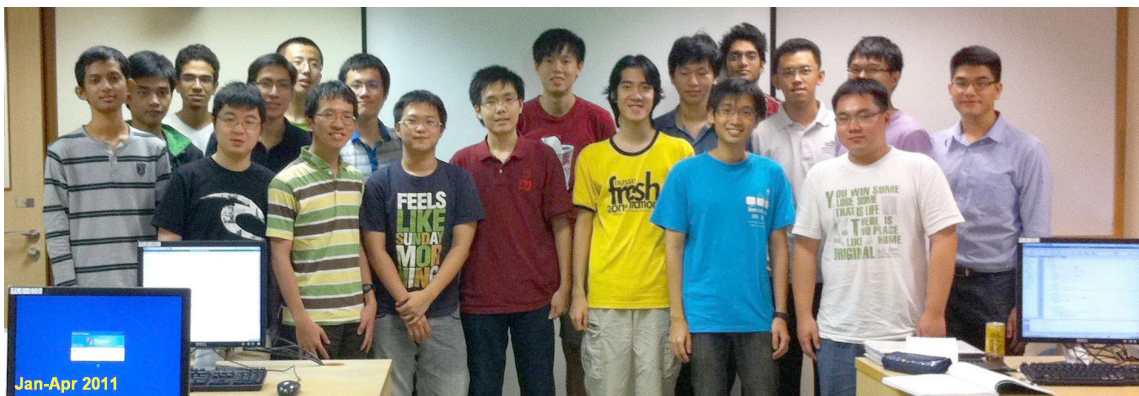
- the proof readers: Seven of CS3233 students above (underlined) plus Tay Wenbin.

- Last but not least, I want to re-thank my wife, Grace Suryani, for letting me do another round of tedious book editing process while she was pregnant with our first baby: Jane Angelina Halim.

Acknowledgments for the Third Edition

From Steven: **Again, I want to thank**

- the $\approx$ 2000 buyers of the 2nd edition as of 24 May 2013 (this number is no longer updated). Thanks :).

- fellow Teaching Assistant of CS3233 @ NUS in the past two years: Harta Wijaya, Trinh Tuan Phuong, and Huang Da.

- my CS3233 students in Sem2 AY2011/2012 who contributed in both technical and presentation aspects of the second edition of this book, in alphabetical order: Cao Sheng, Chua Wei Kuan, Han Yu, Huang Da, Huynh Ngoc Tai, Ivan Reinaldo, John Goh Choo Ern, Le Viet Tien, Lim Zhi Qin, Nalin Ilango, Nguyen Hoang Duy, Nguyen Phi Long, Nguyen Quoc Phong, Pallav Shinghal, Pan Zhengyang, Pang Yan Han, Song Yangyu, Tan Cheng Yong Desmond, Tay Wenbin, Yang Mansheng, Zhao Yang, Zhou Yiming, and two other students who prefer to be anonymous.

- the proof readers: Six of CS3233 students in Sem2 AY2011/2012 (underlined) and Hubert Teo Hua Kian.

- my CS3233 students in Sem2 AY2012/2013 who contributed in both technical and presentation aspects of the second edition of this book, in alphabetical order: Arnold Christopher Koroa, Cao Luu Quang, Lim Puay Ling Pauline, Erik Alexander Qvick Faxaa, Jonathan Darryl Widjaja, Nguyen Tan Sy Nguyen, Nguyen Truong Duy, Ong Ming Hui, Pan Yuxuan, Shubham Goyal, Sudhanshu Khemka, Tang Binbin, Trinh Ngoc Khanh, Yao Yujian, Zhao Yue, and Zheng Naijia.

Jan-Apr 2013

- the NUS Centre for Development of Teaching and Learning (CDTL) for giving the initial funding to build the algorithm visualization website.

- my wife Grace Suryani and my daughter Jane Angelina for your love in our family.

To a better future of humankind,
STEVEN and FELIX HALIM
Singapore, 24 May 2013

Copyright

Authors' Profiles

Steven Halim, PhD[1]

stevenhalim@gmail.com

Steven Halim is currently a lecturer in School of Computing, National University of Singapore (SoC, NUS). He teaches several programming courses in NUS, ranging from basic programming methodology, intermediate data structures and algorithms, and also the 'Competitive Programming' module that uses this book. He is the coach of both the NUS ACM ICPC teams and the Singapore IOI team. He participated in several ACM ICPC Regional as student (Singapore 2001, Aizu 2003, Shanghai 2004). So far, he and other trainers @ NUS have successfully groomed two ACM ICPC World Finalist teams (2009-2010; 2012-2013) as well as two gold, six silver, and seven bronze IOI medalists (2009-2012).

Steven is happily married with Grace Suryani Tioso and currently has one daughter: Jane Angelina Halim.

Felix Halim, PhD[2]

felix.halim@gmail.com

Felix Halim now holds a PhD degree from SoC, NUS. In terms of programming contests, Felix has a much more colourful reputation than his older brother. He was IOI 2002 contestant (representing Indonesia). His ICPC teams (at that time, Bina Nusantara University) took part in ACM ICPC Manila Regional 2003-2004-2005 and obtained rank 10th, 6th, and 10th respectively. Then, in his final year, his team finally won ACM ICPC Kaohsiung Regional 2006 and thus became ACM ICPC World Finalists @ Tokyo 2007 (44th place). Today, he actively joins TopCoder Single Round Matches and his highest rating is a **yellow** coder. He now works at Google, Mountain View, United States of America.

---

[1]PhD Thesis: "An Integrated White+Black Box Approach for Designing and Tuning Stochastic Local Search Algorithms", 2009.

[2]PhD Thesis: "Solving Big Data Problems: from Sequences to Tables and Graphs", 2012.

Abbreviations

A* : A Star
ACM : Assoc of Computing Machinery
AC : Accepted
APSP : All-Pairs Shortest Paths
AVL : Adelson-Velskii Landis (BST)

BNF : Backus Naur Form
BFS : Breadth First Search
BI : Big Integer
BIT : Binary Indexed Tree
BST : Binary Search Tree

CC : Coin Change
CCW : Counter ClockWise
CF : Cumulative Frequency
CH : Convex Hull
CS : Computer Science
CW : ClockWise

DAG : Directed Acyclic Graph
DAT : Direct Addressing Table
D&C : Divide and Conquer
DFS : Depth First Search
DLS : Depth Limited Search
DP : Dynamic Programming
DS : Data Structure

ED : Edit Distance

FIFO : First In First Out
FT : Fenwick Tree

GCD : Greatest Common Divisor

ICPC : Intl Collegiate Prog Contest
IDS : Iterative Deepening Search
IDA* : Iterative Deepening A Star
IOI : Intl Olympiad in Informatics
IPSC : Internet Problem Solving Contest

LA : Live Archive [33]
LCA : Lowest Common Ancestor
LCM : Least Common Multiple
LCP : Longest Common Prefix
LCS$_1$: Longest Common Subsequence
LCS$_2$: Longest Common Substring
LIFO : Last In First Out
LIS : Longest Increasing Subsequence
LRS : Longest Repeated Substring

LSB : Least Significant Bit

MCBM : Max Cardinality Bip Matching
MCM : Matrix Chain Multiplication
MCMF : Min-Cost Max-Flow
MIS : Maximum Independent Set
MLE : Memory Limit Exceeded
MPC : Minimum Path Cover
MSB : Most Significant Bit
MSSP : Multi-Sources Shortest Paths
MST : Minimum Spanning Tree
MWIS : Max Weighted Independent Set
MVC : Minimum Vertex Cover

OJ : Online Judge

PE : Presentation Error

RB : Red-Black (BST)
RMQ : Range Min (or Max) Query
RSQ : Range Sum Query
RTE : Run Time Error

SSSP : Single-Source Shortest Paths
SA : Suffix Array
SPOJ : Sphere Online Judge
ST : Suffix Tree
STL : Standard Template Library

TLE : Time Limit Exceeded

USACO : USA Computing Olympiad
UVa : University of Valladolid [47]

WA : Wrong Answer
WF : World Finals

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Competitive Programming

The core directive in 'Competitive Programming' is this: "Given well-known Computer Science (CS) problems, solve them as quickly as possible!".

Let's digest the terms one by one. The term 'well-known CS problems' implies that in competitive programming, we are dealing with *solved* CS problems and *not* research problems (where the solutions are still unknown). Some people (at least the problem author) have definitely solved these problems before. To 'solve them' implies that we[1] must push our CS knowledge to a certain required level so that we can produce working code that can solve these problems too—at least in terms of getting the *same* output as the problem author using the problem author's secret[2] test data within the stipulated time limit. The need to solve the problem 'as quickly as possible' is where the competitive element lies—speed is a very natural goal in human behavior.

An illustration: UVa Online Judge [47] Problem Number 10911 (Forming Quiz Teams).

Abridged Problem Description:

Let (x, y) be the coordinates of a student's house on a 2D plane. There are $2N$ students and we want to **pair** them into N groups. Let d_i be the distance between the houses of 2 students in group i. Form N groups such that $cost = \sum_{i=1}^{N} d_i$ is **minimized**. Output the minimum *cost*. Constraints: $1 \le N \le 8$ and $0 \le x, y \le 1000$.

Sample input:
$N = 2$; Coordinates of the $2N = 4$ houses are $\{1, 1\}$, $\{8, 6\}$, $\{6, 8\}$, and $\{1, 3\}$.

Sample output:
$cost = 4.83$.

Can you solve this problem?
If so, how many minutes would you likely require to complete the working code?
Think and try not to flip this page immediately!

---

[1] Some programming competitions are done in a team setting to encourage teamwork as software engineers usually do not work alone in real life.

[2] By hiding the actual test data from the problem statement, competitive programming encourages the problem solvers to exercise their mental strength to think of all possible corner cases of the problem and test their programs with those cases. This is typical in real life where software engineers have to test their software a lot to make sure that the software meets the requirements set by clients.
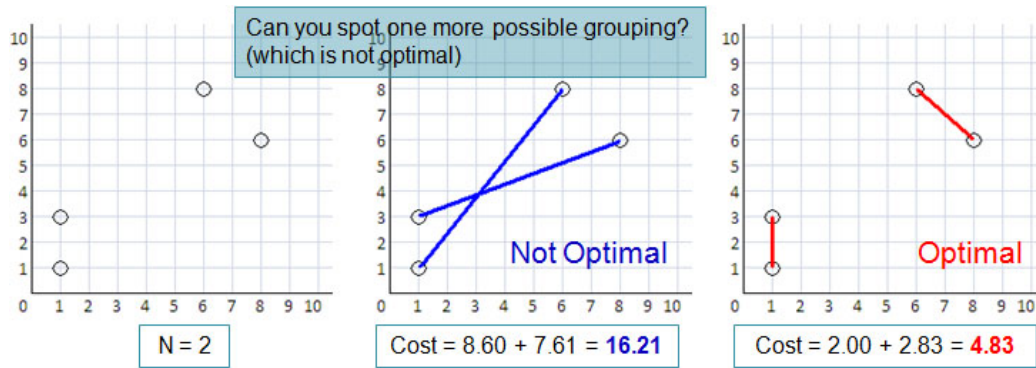
Figure 1.1: Illustration of UVa 10911 - Forming Quiz Teams

Now ask yourself: Which of the following best describes you? Note that if you are unclear with the material or the terminology shown in this chapter, you can re-read it again after going through this book once.

- Uncompetitive programmer A (a.k.a. the blurry one):
  Step 1: Reads the problem and becomes confused. (This problem is new for him).
  Step 2: Tries to code something: Reading the non-trivial input and output.
  Step 3: Realizes that all his attempts are *not* **Accepted (AC)**:
  Greedy (Section 3.4): Repeatedly pairing the two remaining students with the shortest separating distances gives the **Wrong Answer (WA)**.
  Naïve **Complete Search**: Using recursive backtracking (Section 3.2) and trying all possible pairings yields **Time Limit Exceeded (TLE)**.

- Uncompetitive programmer B (Give up):
  Step 1: Reads the problem and realizes that he has seen this problem before. But also remembers that he has not learned how to solve this kind of problem... He is not aware of the **Dynamic Programming (DP)** solution (Section 3.5)...
  Step 2: Skips the problem and reads another problem in the problem set.

- (Still) Uncompetitive programmer C (Slow):
  Step 1: Reads the problem and realizes that it is a hard problem: **'minimum weight perfect matching on a small general weighted graph'**. However, since the input size is small, this problem is solvable using DP. The DP state is a **bitmask** that describes a matching status, and matching unmatched students i and j will turn on two bits i and j in the bitmask (Section 8.3.1).
  Step 2: Codes I/O routine, writes recursive top-down DP, tests, **debugs >.<**...
  Step 3: *After 3 hours*, his solution obtains AC (passed all the secret test data).

- Competitive programmer D:
  Completes all the steps taken by uncompetitive programmer C in $\leq$ 30 minutes.

- Very competitive programmer E:
  A very competitive programmer (e.g. the red 'target' coders in TopCoder [32]) would solve this 'well known' problem $\leq$ 15 minutes...

Please note that being well-versed in competitive programming is *not* the end goal, but only a means to an end. The true end goal is to produce all-rounder computer scientists/programmers who are much readier to produce better software and to face harder CS research problems in the future. The founders of the ACM International Collegiate Programming Contest (ICPC) [66] have this vision and we, the authors, agree with it. With this book, we play our little role in preparing the current and the future generations to be more competitive in dealing with well-known CS problems frequently posed in the recent ICPCs and the International Olympiad in Informatics (IOI)s.

Exercise 1.1.1: The greedy strategy of the uncompetitive programmer A above actually works for the sample test case shown in Figure 1.1. Please give a *better* counter example!

Exercise 1.1.2: Analyze the time complexity of the naïve complete search solution by uncompetitive programmer A above to understand why it receives the TLE verdict!

Exercise 1.1.3*: Actually, a clever recursive backtracking solution *with pruning* can still solve this problem. Solve this problem without using a DP table!

1.2   Tips to be Competitive

If you strive to be like competitive programmers D or E as illustrated above—that is, if you want to be selected (via provincial/state → national team selections) to participate and obtain a medal in the IOI [34], or to be one of the team members that represents your University in the ACM ICPC [66] (nationals → regionals → and up to world finals), or to do well in other programming contests—then this book is definitely for you!

In the subsequent chapters, you will learn everything from the basic to the intermediate or even to the advanced[3] data structures and algorithms that have frequently appeared in recent programming contests, compiled from many sources [50, 9, 56, 7, 40, 58, 42, 60, 1, 38, 8, 59, 41, 62, 46] (see Figure 1.4). You will not only learn the concepts behind the data structures and algorithms, but also how to implement them efficiently and apply them to appropriate contest problems. On top of that, you will also learn many programming tips derived from our own experiences that can be helpful in contest situations. We start this book by giving you several general tips below:

1.2.1   Tip 1: Type Code Faster!

No kidding! Although this tip may not mean much as ICPC and (especially) IOI are not typing contests, we have seen Rank i and Rank $i + 1$ ICPC teams separated only by a few minutes and frustrated IOI contestants who miss out on salvaging important marks by not being able to code a last-minute brute force solution properly. When you can solve the same number of problems as your competitor, it will then be down to coding skill (your ability to produce concise and robust code) and ... typing speed.

Try this typing test at http://www.typingtest.com and follow the instructions there on how to improve your typing skill. Steven's is ∼85-95 wpm and Felix's is ∼55-65 wpm. If your typing speed is much less than these numbers, please take this tip seriously!

On top of being able to type alphanumeric characters quickly and correctly, you will also need to familiarize your fingers with the positions of the frequently used programming language characters: parentheses () or {} or square brackets [] or angle brackets <>, the semicolon ; and colon :, single quotes '' for characters, double quotes "" for strings, the ampersand &, the vertical bar or the 'pipe' |, the exclamation mark !, etc.

As a little practice, try typing the C++ source code below as fast as possible.

```
#include <algorithm>            // if you have problems with this C++ code,
#include <cmath>             // consult your programming text books first...
#include <cstdio>
#include <cstring>
using namespace std;
```

---

[3]Whether you perceive the material presented in this book to be of intermediate or advanced difficulty depends on your programming skill prior to reading this book.

```
         /* Forming Quiz Teams, the solution for UVa 10911 above */
            // using global variables is a bad software engineering practice,
int N, target;                    // but it is OK for competitive programming
double dist[20][20], memo[1 << 16];   // 1 << 16 = 2^16, note that max N = 8

double matching(int bitmask) {                       // DP state = bitmask
                        // we initialize 'memo' with -1 in the main function
  if (memo[bitmask] > -0.5)            // this state has been computed before
    return memo[bitmask];                      // simply lookup the memo table
  if (bitmask == target)              // all students are already matched
    return memo[bitmask] = 0;                              // the cost is 0

  double ans = 2000000000.0;                  // initialize with a large value
  int p1, p2;
  for (p1 = 0; p1 < 2 * N; p1++)
    if (!(bitmask & (1 << p1)))
      break;                          // find the first bit that is off
  for (p2 = p1 + 1; p2 < 2 * N; p2++)              // then, try to match p1
    if (!(bitmask & (1 << p2)))     // with another bit p2 that is also off
      ans = min(ans,                               // pick the minimum
            dist[p1][p2] + matching(bitmask | (1 << p1) | (1 << p2)));

  return memo[bitmask] = ans;     // store result in a memo table and return
}

int main() {
  int i, j, caseNo = 1, x[20], y[20];
  // freopen("10911.txt", "r", stdin);      // redirect input file to stdin

  while (scanf("%d", &N), N) {                       // yes, we can do this :)
    for (i = 0; i < 2 * N; i++)
      scanf("%*s %d %d", &x[i], &y[i]);              // '%*s' skips names
    for (i = 0; i < 2 * N - 1; i++)        // build pairwise distance table
      for (j = i + 1; j < 2 * N; j++)      // have you used 'hypot' before?
        dist[i][j] = dist[j][i] = hypot(x[i] - x[j], y[i] - y[j]);

    // use DP to solve min weighted perfect matching on small general graph
    for (i = 0; i < (1 << 16); i++) memo[i] = -1.0;  // set -1 to all cells
    target = (1 << (2 * N)) - 1;
    printf("Case %d: %.2lf\n", caseNo++, matching(0));
} } // return 0;
```

For your reference, the explanation of this 'Dynamic Programming with bitmask' solution is given in Section 2.2, 3.5, and 8.3.1. Do not be alarmed if you do not understand it yet.

1.2.2 Tip 2: Quickly Identify Problem Types

In ICPCs, the contestants (teams) are given a **set** of problems ($\approx$ 7-12 problems) of varying types. From our observation of recent ICPC Asia Regional problem sets, we can categorize the problems types and their rate of appearance as in Table 1.1.

In IOIs, the contestants are given 6 tasks over 2 days (8 tasks over 2 days in 2009-2010) that cover items 1-5 and 10, with a *much smaller* subset of items 6-10 in Table 1.1. For details, please refer to the 2009 IOI syllabus [20] and the IOI 1989-2008 problem classification [67].

No	Category	In This Book	Frequency
1.	Ad Hoc	Section 1.4	1-2
2.	Complete Search (Iterative/Recursive)	Section 3.2	1-2
3.	Divide and Conquer	Section 3.3	0-1
4.	Greedy (usually the original ones)	Section 3.4	0-1
5.	Dynamic Programming (usually the original ones)	Section 3.5	1-3
6.	Graph	Chapter 4	1-2
7.	Mathematics	Chapter 5	1-2
8.	String Processing	Chapter 6	1
9.	Computational Geometry	Chapter 7	1
10.	Some Harder/Rare Problems	Chapter 8-9	1-2
		Total in Set	8-17 ($\approx\,\leq 12$)

Table 1.1: Recent ACM ICPC (Asia) Regional Problem Types

The classification in Table 1.1 is adapted from [48] and by no means complete. Some techniques, e.g. 'sorting', are not classified here as they are 'trivial' and usually used only as a 'sub-routine' in a bigger problem. We do not include 'recursion' as it is embedded in categories like recursive backtracking or Dynamic Programming. We also omit 'data structures' as the usage of efficient data structure can be considered to be integral for solving harder problems. Of course, problems sometimes require mixed techniques: A problem can be classified into more than one type. For example, Floyd Warshall's algorithm is both a solution for the All-Pairs Shortest Paths (APSP, Section 4.5) graph problem and a Dynamic Programming (DP) algorithm (Section 3.5). Prim's and Kruskal's algorithms are both solutions for the Minimum Spanning Tree (MST, Section 4.3) graph problem and Greedy algorithms (Section 3.4). In Section 8.4, we will discuss (harder) problems that require more than one algorithms and/or data structures to be solved.

In the (near) future, these classifications may change. One significant example is Dynamic Programming. This technique was not known before 1940s, nor frequently used in ICPCs or IOIs before mid 1990s, but it is considered a definite prerequisite today. As an illustration: There were ≥ 3 DP problems (out of 11) in the recent ICPC World Finals 2010.

However, the main goal is *not* just to associate problems with the techniques required to solve them like in Table 1.1. Once you are familiar with most of the topics in this book, you should also be able to classify problems into the three types in Table 1.2.

No	Category	Confidence and Expected Solving Speed
A.	I have solved this type before	I am sure that I can re-solve it again (and fast)
B.	I have seen this type before	But that time I know I cannot solve it yet
C.	I have not seen this type before	See discussion below

Table 1.2: Problem Types (Compact Form)

To be *competitive*, that is, *do well* in a programming contest, you must be able to confidently and frequently classify problems as type A and minimize the number of problems that you classify into type B. That is, you need to acquire sufficient algorithm knowledge and develop your programming skills so that you consider many classical problems to be easy. However, to *win* a programming contest, you will also need to develop sharp *problem solving skills* (e.g. reducing the given problem to a known problem, identifying subtle hints or special

properties in the problem, attacking the problem from a non obvious angle, etc) so that you (or your team) will be able to derive the required solution to a hard/original type C problem in IOI or ICPC Regionals/World Finals and do so *within* the duration of the contest.

UVa	Title	Problem Type	Hint
10360	Rat Attack	Complete Search or DP	Section 3.2
10341	Solve It		Section 3.3
11292	Dragon of Loowater		Section 3.4
11450	Wedding Shopping		Section 3.5
10911	Forming Quiz Teams	DP with bitmask	Section 8.3.1
11635	Hotel Booking		Section 8.4
11506	Angry Programmer		Section 4.6
10243	Fire! Fire!! Fire!!!		Section 4.7.1
10717	Mint		Section 8.4
11512	GATTACA		Section 6.6
10065	Useless Tile Packers		Section 7.3.7

Table 1.3: Exercise: Classify These UVa Problems

Exercise 1.2.1: Read the UVa [47] problems shown in Table 1.3 and determine their problem types. Two of them have been identified for you. Filling this table is easy after mastering this book—all the techniques required to solve these problems are discussed in this book.

1.2.3 Tip 3: Do Algorithm Analysis

Once you have designed an algorithm to solve a particular problem in a programming contest, you must then ask this question: Given the maximum input bound (usually given in a good problem description), can the currently developed algorithm, with its time/space complexity, pass the time/memory limit given for that particular problem?

Sometimes, there are more than one way to attack a problem. Some approaches may be incorrect, others not fast enough, and yet others 'overkill'. A good strategy is to brainstorm for many possible algorithms and then pick the **simplest solution that works** (i.e. is fast enough to pass the time and memory limit and yet still produce the correct answer)[4]!

Modern computers are quite fast and can process[5] up to $\approx 100M$ (or 10^8; $1M = 1,000,000$) operations in a few seconds. You can use this information to determine if your algorithm will run in time. For example, if the maximum input size n is $100K$ (or 10^5; $1K = 1,000$), and your current algorithm has a time complexity of $O(n^2)$, common sense (or your calculator) will inform you that $(100K)^2$ or 10^{10} is a very large number that indicates that your algorithm will require (on the order of) hundreds of seconds to run. You will thus need to devise a faster (and also correct) algorithm to solve the problem. Suppose you find one that runs with a time complexity of $O(n \log_2 n)$. Now, your calculator will inform you that $10^5 \log_2 10^5$ is just 1.7×10^6 and common sense dictates that the algorithm (which should now run in under a second) will likely be able to pass the time limit.

---

[4]Discussion: It is true that in programming contests, picking the simplest algorithm that works is crucial for doing well in that programming contest. However, during *training sessions*, where time constraints are not an issue, it can be beneficial to spend more time trying to solve a certain problem using the *best possible algorithm*. We are better prepared this way. If we encounter a harder version of the problem in the future, we will have a greater chance of obtaining and implementing the correct solution!

[5]Treat this as a rule of thumb. This numbers may vary from machine to machine.

The problem bounds are as important as your algorithm's time complexity in determining if your solution is appropriate. Suppose that you can only devise a relatively-simple-to-code algorithm that runs with a horrendous time complexity of $O(n^4)$. This may appear to be an infeasible solution, but if $n \leq 50$, then you have actually solved the problem. You can implement your $O(n^4)$ algorithm with impunity since 50^4 is just $6.25M$ and your algorithm should still run in around a second.

Note, however, that the order of complexity does not necessarily indicate the actual number of operations that your algorithm will require. If each iteration involves a large number of operations (many floating point calculations, or a significant number of constant sub-loops), or if your implementation has a high 'constant' in its execution (unnecessarily repeated loops or multiple passes, or even I/O or execution overhead), your code may take longer to execute than expected. However, this will usually not be the case as the problem authors should have designed the time limits so that a well-coded algorithm with a suitable time complexity will achieve an AC verdict.

By analyzing the complexity of your algorithm with the given input bound and the stated time/memory limit, you can better decide whether you should attempt to implement your algorithm (which will take up precious time in the ICPCs and IOIs), attempt to improve your algorithm first, or switch to other problems in the problem set.

As mentioned in the preface of this book, we will *not* discuss the concept of algorithmic analysis in details. We *assume* that you already have this basic skill. There are a multitude of other reference books (for example, the "Introduction to Algorithms" [7], "Algorithm Design" [38], "Algorithms" [8], etc) that will help you to understand the following prerequisite concepts/techniques in algorithmic analysis:

- Basic time and space complexity analysis for iterative and recursive algorithms:

  - An algorithm with k-nested loops of about n iterations each has $O(n^k)$ complexity.

  - If your algorithm is recursive with b recursive calls per level and has L levels, the algorithm has roughly $O(b^L)$ complexity, but this is a only a rough upper bound. The actual complexity depends on what actions are done per level and whether pruning is possible.

  - A Dynamic Programming algorithm or other iterative routine which processes a 2D $n \times n$ matrix in $O(k)$ per cell runs in $O(k \times n^2)$ time. This is explained in further detail in Section 3.5.

- More advanced analysis techniques:

  - Prove the correctness of an algorithm (especially for Greedy algorithms in Section 3.4), to minimize your chance of getting the 'Wrong Answer' verdict.

  - Perform the amortized analysis (e.g. see Chapter 17 of [7])—although rarely used in contests—to minimize your chance of getting the 'Time Limit Exceeded' verdict, or worse, considering your algorithm to be too slow and skips the problem when it is in fact fast enough in amortized sense.

  - Do output-sensitive analysis to analyze algorithm which (also) depends on output size and minimize your chance of getting the 'Time Limit Exceeded' verdict. For example, an algorithm to search for a string with length m in a long string with the help of a Suffix Tree (that is already built) runs in $O(m+occ)$ time. The time taken for this algorithm to run depends not only on the input size m but also the output size—the number of occurrences occ (see more details in Section 6.6).

- Familiarity with these bounds:
  - $2^{10} = 1,024 \approx 10^3$, $2^{20} = 1,048,576 \approx 10^6$.
  - 32-bit signed integers (`int`) and 64-bit signed integers (`long long`) have upper limits of $2^{31} - 1 \approx 2 \times 10^9$ (safe for up to ≈ 9 decimal digits) and $2^{63} - 1 \approx 9 \times 10^{18}$ (safe for up to ≈ 18 decimal digits) respectively.
  - Unsigned integers can be used if only non-negative numbers are required. 32-bit unsigned integers (`unsigned int`) and 64-bit unsigned integers (`unsigned long long`) have upper limits of $2^{32} - 1 \approx 4 \times 10^9$ and $2^{64} - 1 \approx 1.8 \times 10^{19}$ respectively.
  - If you need to store integers $\geq 2^{64}$, use the Big Integer technique (Section 5.3).
  - There are $n!$ permutations and 2^n subsets (or combinations) of n elements.
  - The best time complexity of a comparison-based sorting algorithm is $\Omega(n \log_2 n)$.
  - Usually, $O(n \log_2 n)$ algorithms are sufficient to solve most contest problems.
  - The largest input size for typical programming contest problems must be $< 1M$. Beyond that, the time needed to read the input (the Input/Output routine) will be the bottleneck.
  - A typical year 2013 CPU can process $100M = 10^8$ operations in a few seconds.

Many novice programmers would skip this phase and immediately begin implementing the first (naïve) algorithm that they can think of only to realize that the chosen data structure or algorithm is not efficient enough (or wrong). Our advice for ICPC contestants[6]: Refrain from coding until you are sure that your algorithm is both correct and fast enough.

n	Worst AC Algorithm	Comment
$\leq [10..11]$	$O(n!), O(n^6)$	e.g. Enumerating permutations (Section 3.2)
$\leq [15..18]$	$O(2^n \times n^2)$	e.g. DP TSP (Section 3.5.2)
$\leq [18..22]$	$O(2^n \times n)$	e.g. DP with bitmask technique (Section 8.3.1)
≤ 100	$O(n^4)$	e.g. DP with 3 dimensions + $O(n)$ loop, $_nC_{k=4}$
≤ 400	$O(n^3)$	e.g. Floyd Warshall's (Section 4.5)
$\leq 2K$	$O(n^2 \log_2 n)$	e.g. 2-nested loops + a tree-related DS (Section 2.3)
$\leq 10K$	$O(n^2)$	e.g. Bubble/Selection/Insertion Sort (Section 2.2)
$\leq 1M$	$O(n \log_2 n)$	e.g. Merge Sort, building Segment Tree (Section 2.3)
$\leq 100M$	$O(n), O(\log_2 n), O(1)$	Most contest problem has $n \leq 1M$ (I/O bottleneck)

Table 1.4: Rule of thumb time complexities for the 'Worst AC Algorithm' for various single-test-case input sizes n, assuming that your CPU can compute $100M$ items in 3s.

To help you understand the growth of several common time complexities, and thus help you judge how fast is 'enough', refer to Table 1.4. Variants of such tables are also found in many other books on data structures and algorithms. This table is written from a *programming contestant's perspective*. Usually, the input size constraints are given in a (good) problem description. With the assumption that a typical CPU can execute a hundred million operations in around 3 seconds (the typical time limit in most UVa [47] problems), we can predict the 'worst' algorithm that can still pass the time limit. Usually, the simplest algorithm has the poorest time complexity, but if it can pass the time limit, just use it!

---

[6]Unlike ICPC, the IOI tasks can usually be solved (partially or fully) using several possible solutions, each with different time complexities and subtask scores. To gain valuable points, it may be good to use a brute force solution to score a few points and to understand the problem better. There will be no significant time penalty as IOI is not a speed contest. Then, iteratively improve the solution to gain more points.

From Table 1.4, we see the importance of using good algorithms with small orders of growth as they allow us to solve problems with larger input sizes. But a faster algorithm is usually non-trivial and sometimes substantially harder to implement. In Section 3.2.3, we discuss a few tips that may allow the same class of algorithms to be used with larger input sizes. In subsequent chapters, we also explain efficient algorithms for various computing problems.

Exercise 1.2.2: Please answer the following questions below using your current knowledge about classic algorithms and their time complexities. After you have finished reading this book once, it may be beneficial to attempt this exercise again.

1. There are n webpages ($1 \leq n \leq 10M$). Each webpage i has a page rank r_i. You want to pick the top 10 pages with the highest page ranks. Which method is better?

   (a) Load all n webpages' page rank to memory, sort (Section 2.2) them in descending page rank order, obtaining the top 10.

   (b) Use a priority queue data structure (a heap) (Section 2.3).

2. Given an $M \times N$ integer matrix Q ($1 \leq M, N \leq 30$), determine if there exists a sub-matrix of Q of size $A \times B$ ($1 \leq A \leq M, 1 \leq B \leq N$) where $\texttt{mean}(Q) = 7$.

   (a) Try all possible sub-matrices and check if the mean of each sub-matrix is 7. This algorithm runs in $O(M^3 \times N^3)$.

   (b) Try all possible sub-matrices, but in $O(M^2 \times N^2)$ with this technique: _____ .

3. Given a list L with $10K$ integers, you need to *frequently* obtain $\texttt{sum(i, j)}$, i.e. the sum of $\texttt{L[i]} + \texttt{L[i+1]} + \texttt{...+ L[j]}$. Which data structure should you use?

   (a) Simple Array (Section 2.2).

   (b) Simple Array pre-processed with Dynamic Programming (Section 2.2 & 3.5).

   (c) Balanced Binary Search Tree (Section 2.3).

   (d) Binary Heap (Section 2.3).

   (e) Segment Tree (Section 2.4.3).

   (f) Binary Indexed (Fenwick) Tree (Section 2.4.4).

   (g) Suffix Tree (Section 6.6.2) or its alternative, Suffix Array (Section 6.6.4).

4. Given a set S of N points *randomly* scattered on a 2D plane ($2 \leq N \leq 1000$), find two points $\in S$ that have the greatest separating Euclidean distance. Is an $O(N^2)$ complete search algorithm that tries all possible pairs feasible?

   (a) Yes, such complete search is possible.

   (b) No, we must find another way. We must use: _____ .

5. You have to compute the shortest path between two vertices on a weighted Directed Acyclic Graph (DAG) with $|V|, |E| \leq 100K$. Which algorithm(s) can be used in a typical programming contest (that is, with a time limit of approximately 3 seconds)?

   (a) Dynamic Programming (Section 3.5, 4.2.5, & 4.7.1).

   (b) Breadth First Search (Section 4.2.2 & 4.4.2).

   (c) Dijkstra's (Section 4.4.3).

    (d) Bellman Ford's (Section 4.4.4).

    (e) Floyd Warshall's (Section 4.5).

6. Which algorithm produces a list of the first $10K$ prime numbers with the best time complexity? (Section 5.5.1)

    (a) Sieve of Eratosthenes (Section 5.5.1).

    (b) For each number $i \in$ `[1..10K]`, test if `isPrime(i)` is true (Section 5.5.1).

7. You want to test if the factorial of n, i.e. $n!$ is divisible by an integer m. $1 \le n \le 10000$. What should you do?

    (a) Test if $n! \% m == 0$.

    (b) The naïve approach above will not work, use: _____ (Section 5.5.1).

8. Question 4, but with a larger set of points: $N \le 1M$ and one additional constraint: The points are *randomly scattered* on a 2D plane.

    (a) The complete search mentioned in question 3 can still be used.

    (b) The naïve approach above will not work, use: _____ (Section 7.3.7).

9. You want to enumerate all occurrences of a substring P (of length m) in a (long) string T (of length n), if any. Both n and m have a maximum of 1M characters.

    (a) Use the following C++ code snippet:

```
for (int i = 0; i < n; i++) {
  bool found = true;
  for (int j = 0; j < m && found; j++)
    if (i + j >= n || P[j] != T[i + j]) found = false;
  if (found) printf("P is found at index %d in T\n", i);
}
```

    (b) The naïve approach above will not work, use: _____ (Section 6.4 or 6.6).

1.2.4 Tip 4: Master Programming Languages

There are several programming languages supported in ICPC[7], including C/C++ and Java. Which programming languages should one aim to master?

    Our experience gives us this answer: We prefer C++ with its built-in Standard Template Library (STL) but we still need to master Java. Even though it is slower, Java has powerful built-in libraries and APIs such as BigInteger/BigDecimal, GregorianCalendar, Regex, etc. Java programs are easier to debug with the virtual machine's ability to provide a stack trace

---

[7]Personal opinion: According to the latest IOI 2012 competition rules, Java is currently still not supported in IOI. The programming languages allowed in IOI are C, C++, and Pascal. On the other hand, the ICPC World Finals (and thus most Regionals) allows C, C++ and Java to be used in the contest. Therefore, it is seems that the 'best' language to master is C++ as it is supported in both competitions and it has strong STL support. If IOI contestants choose to master C++, they will have the benefit of being able to use the same language (with an increased level of mastery) for ACM ICPC in their University level pursuits.

when it crashes (as opposed to core dumps or segmentation faults in C/C++). On the other hand, C/C++ has its own merits as well. Depending on the problem at hand, either language may be the better choice for implementing a solution in the shortest time.

Suppose that a problem requires you to compute 25! (the factorial of 25). The answer is very large: 15,511,210,043,330,985,984,000,000. This far exceeds the largest built-in primitive integer data type (unsigned long long: $2^{64} - 1$). As there is no built-in arbitrary-precision arithmetic library in C/C++ as of yet, we would have needed to implement one from scratch. The Java code, however, is exceedingly simple (more details in Section 5.3). In this case, using Java definitely makes for shorter coding time.

```java
import java.util.Scanner;
import java.math.BigInteger;

class Main {                              // standard Java class name in UVa OJ
  public static void main(String[] args) {
    BigInteger fac = BigInteger.ONE;
    for (int i = 2; i <= 25; i++)
      fac = fac.multiply(BigInteger.valueOf(i));   // it is in the library!
    System.out.println(fac);
} }
```

Mastering and understanding the full capability of your favourite programming language is also important. Take this problem with a non-standard input format: the first line of input is an integer N. This is followed by N lines, each starting with the character '0', followed by a dot '.', then followed by an unknown number of digits (up to 100 digits), and finally terminated with three dots '...'.

```
3
0.1227...
0.517611738...
0.734123122344434438992389927...
```

One possible solution is as follows:

```cpp
#include <cstdio>
using namespace std;

int N;          // using global variables in contests can be a good strategy
char x[110];   // make it a habit to set array size a bit larger than needed

int main() {
  scanf("%d\n", &N);
  while (N--) {                        // we simply loop from N, N-1, N-2, ..., 0
    scanf("0.%[0-9]...\n", &x);   // '&' is optional when x is a char array
                        // note: if you are surprised with the trick above,
                    // please check scanf details in www.cppreference.com
    printf("the digits are 0.%s\n", x);
} } // return 0;
```

Source code: `ch1_01_factorial.java`; `ch1_02_scanf.cpp`

Not many C/C++ programmers are aware of partial regex capabilities built into the C standard I/O library. Although `scanf/printf` are C-style I/O routines, they can still be used in C++ code. Many C++ programmers 'force' themselves to use `cin/cout` all the time even though it is sometimes not as flexible as `scanf/printf` and is also far slower.

In programming contests, especially ICPCs, coding time should *not* be the primary bottleneck. Once you figure out the 'worst AC algorithm' that will pass the given time limit, you are expected to be able to translate it into a bug-free code and fast!

Now, try some of the exercises below! If you need more than 10 lines of code to solve any of them, you should revisit and update your knowledge of your programming language(s)! A mastery of the programming languages you use and their built-in routines is extremely important and will help you a lot in programming contests.

Exercise 1.2.3: Produce working code that is *as concise as possible* for the following tasks:

1. Using **Java**, read in a double
   (e.g. `1.4732`, `15.324547327`, etc.)
   echo it, but with a minimum field width of 7 and 3 digits after the decimal point
   (e.g. `ss1.473` (where 's' denotes a space), `s15.325`, etc.)

2. Given an integer n ($n \leq 15$), print π to n digits after the decimal point (rounded).
   (e.g. for $n = 2$, print `3.14`; for $n = 4$, print `3.1416`; for $n = 5$, prints `3.14159`.)

3. Given a date, determine the day of the week (Monday, ..., Sunday) on that day.
   (e.g. 9 August 2010—the launch date of the first edition of this book—is a Monday.)

4. Given n random integers, print the distinct (unique) integers in sorted order.

5. Given the distinct and valid birthdates of n people as triples (DD, MM, YYYY), order them first by ascending birth months (MM), then by ascending birth dates (DD), and finally by ascending age.

6. Given a list of *sorted* integers L of size up to $1M$ items, determine whether a value v exists in L with no more than 20 comparisons (more details in Section 2.2).

7. Generate all possible permutations of {'A', 'B', 'C', ..., 'J'}, the first $N = 10$ letters in the alphabet (see Section 3.2.1).

8. Generate all possible subsets of {0, 1, 2, ..., N-1}, for $N = 20$ (see Section 3.2.1).

9. Given a string that represents a base X number, convert it to an equivalent string in base Y, $2 \leq X, Y \leq 36$. For example: "FF" in base X = 16 (hexadecimal) is "255" in base $Y_1 = 10$ (decimal) and "11111111" in base $Y_2 = 2$ (binary). See Section 5.3.2.

10. Let's define a 'special word' as a lowercase alphabet followed by two consecutive digits. Given a string, replace all 'special words' of length 3 with 3 stars "***", e.g.
    S = "line: a70 and z72 will be replaced, aa24 and a872 will not"
    should be transformed into:
    S = "line: *** and *** will be replaced, aa24 and a872 will not".

11. Given a *valid* mathematical expression involving '+', '-', '*', '/', '(', and ')' in a single line, evaluate that expression. (e.g. a rather complicated but valid expression `3 + (8 - 7.5) * 10 / 5 - (2 + 5 * 7)` should produce `-33.0` when evaluated with standard operator precedence.)

1.2.5   Tip 5: Master the Art of Testing Code

You thought you nailed a particular problem. You identified its problem type, designed the algorithm for it, verified that the algorithm (with the data structures it uses) will run in time (and within memory limits) by considering the time (and space) complexity, and implemented the algorithm, but your solution is still not Accepted (AC).

Depending on the programming contest, you may or may not get credit for solving the problem partially. In ICPC, you will only get points for a particular problem if your team's code solves **all** the secret test cases for that problem. Other verdicts such as Presentation Error (PE), Wrong Answer (WA), Time Limit Exceeded (TLE), Memory Limit Exceeded (MLE), Run Time Error (RTE), etc. do not increase your team's points. In current IOI (2010-2012), the subtask scoring system is used. Test cases are grouped into subtasks, usually simpler variants of the original task with smaller input bounds. You will only be credited for solving a subtask if your code solves all test cases in it. You are given *tokens* that you can use (sparingly) throughout the contest to view the judge's evaluation of your code.

In either case, you will need to be able to design good, comprehensive and tricky test cases. The sample input-output given in the problem description is by nature trivial and therefore usually not a good means for determining the correctness of your code.

Rather than wasting submissions (and thus accumulating time or score penalties) in ICPC or tokens in IOI, you may want to design tricky test cases for testing your code on your own machine[8]. Ensure that your code is able to solve them correctly (otherwise, there is no point submitting your solution since it is likely to be incorrect—unless you want to test the test data bounds).

Some coaches encourage their students to compete with each other by designing test cases. If student A's test cases can break student B's code, then A will get bonus points. You may want to try this in your team training :).

Here are some guidelines for designing good test cases from our experience. These are typically the steps that have been taken by problem authors.

1. Your test cases should include the sample test cases since the sample output is guaranteed to be correct. Use 'fc' in Windows or 'diff' in UNIX to check your code's output (when given the sample input) against the sample output. Avoid manual comparison as humans are prone to error and are not good at performing such tasks, especially for problems with strict output formats (e.g. blank line *between* test cases versus *after every* test cases). To do this, *copy and paste* the sample input and sample output from the problem description, then save them to files (named as 'input' and 'output' or anything else that is sensible). Then, after compiling your program (let's assume the executable's name is the 'g++' default 'a.out'), execute it with an I/O redirection: './a.out < input > myoutput'. Finally, execute 'diff myoutput output' to highlight any (potentially subtle) differences, if any exist.

2. For problems with multiple test cases in a single run (see Section 1.3.2), you should include two identical sample test cases consecutively in the same run. Both must output the same known correct answers. This helps to determine if you have forgotten to initialize any variables—if the first instance produces the correct answer but the second one does not, it is likely that you have not reset your variables.

3. Your test cases should include tricky corner cases. Think like the problem author and try to come up with the worst possible input for your algorithm by identifying cases

---

[8]Programming contest environments differ from one contest to another. This can disadvantage contestants who rely too much on fancy Integrated Development Environment (IDE) (e.g. Visual Studio, Eclipse, etc) for debugging. It may be a good idea to practice coding with just a **text editor** and a **compiler**!

that are 'hidden' or implied within the problem description. These cases are usually included in the judge's secret test cases but *not* in the sample input and output. Corner cases typically occur at extreme values such as $N = 0$, $N = 1$, negative values, large final (and/or intermediate) values that does not fit 32-bit signed integer, etc.

4. Your test cases should include *large* cases. Increase the input size incrementally up to the maximum input bounds stated in the problem description. Use large test cases with trivial structures that are easy to verify with manual computation and large *random* test cases to test if your code terminates in time and still produces reasonable output (since the correctness would be difficult to verify here). Sometimes your program may work for small test cases, but produces wrong answer, crashes, or exceeds the time limit when the input size increases. If that happens, check for overflows, out of bound errors, or improve your algorithm.

5. Though this is rare in modern programming contests, do not assume that the input will always be nicely formatted if the problem description does not explicitly state it (especially for a badly written problem). Try inserting additional whitespace (spaces, tabs) in the input and test if your code is still able to obtain the values correctly without crashing.

However, after carefully following all these steps, you may still get non-AC verdicts. In ICPC, you (and your team) can actually consider the judge's verdict and the leader board (usually available for the first four hours of the contest) in determining your next course of action. In IOI 2010-2012, contestants have a limited number of tokens to use for checking the correctness of their submitted code against the secret test cases. With more experience in such contests, you will be able to make better judgments and choices.

Exercise 1.2.4: Situational awareness
(mostly applicable in the ICPC setting—this is not as relevant in IOI).

1. You receive a WA verdict for a very easy problem. What should you do?

    (a) Abandon this problem for another.
    (b) Improve the performance of your solution (code optimizations/better algorithm).
    (c) Create tricky test cases to find the bug.
    (d) (In team contests): Ask your team mate to re-do the problem.

2. You receive a TLE verdict for your $O(N^3)$ solution.
   However, the maximum N is just 100. What should you do?

    (a) Abandon this problem for another.
    (b) Improve the performance of your solution (code optimizations/better algorithm).
    (c) Create tricky test cases to find the bug.

3. Follow up to Question 2: What if the maximum N is 100.000?

4. Another follow up to Question 2: What if the maximum N is 1.000, the output only depends on the size of input N, and you still have *four hours* of competition time left?

5. You receive an RTE verdict. Your code (seems to) execute perfectly on your machine. What should you do?

6. Thirty minutes into the contest, you take a glance at the leader board. There are *many* other teams that have solved a problem X that your team has not attempted. What should you do?

7. Midway through the contest, you take a glance at the leader board. The leading team (assume that it is not your team) has just solved problem Y. What should you do?

8. Your team has spent two hours on a nasty problem. You have submitted several implementations by different team members. All submissions have been judged incorrect. You have no idea what's wrong. What should you do?

9. There is one hour to go before the end of the contest. You have 1 WA code and 1 fresh idea for *another* problem. What should you (or your team) do?

    (a) Abandon the problem with the WA code, switch to the other problem in an attempt to solve one more problem.

    (b) Insist that you have to debug the WA code. There is not enough time to start working on a new problem.

    (c) (In ICPC): Print the WA code. Ask two other team members to scrutinize it while you switch to that other problem in an attempt to solve *two* more problems.

1.2.6 Tip 6: Practice and More Practice

Competitive programmers, like real athletes, must train regularly and keep 'programming-fit'. Thus in our second last tip, we provide a list of several websites with resources that can help improve your problem solving skill. We believe that success comes as a result of a continuous effort to better yourself.

The University of Valladolid (UVa, from Spain) Online Judge [47] contains past ACM contest problems (Locals, Regionals, and up to World Finals) plus problems from other sources, including various problems from contests hosted by UVa. You can solve these problems and submit your solutions to the Online Judge. The correctness of your program will be reported as soon as possible. Try solving the problems mentioned in this book and you might see your name on the top-500 authors rank list someday :-).

As of 24 May 2013, one needs to solve ≥ 542 problems to be in the top-500. Steven is ranked 27 (for solving 1674 problems) while Felix is ranked 37 (for solving 1487 problems) out of ≈ 149008 UVa users (and a total of ≈ 4097 problems).

UVa's 'sister' online judge is the ACM ICPC Live Archive [33] that contains *almost all* recent ACM ICPC Regionals and World Final problem sets since year 2000. Train here if you want to do well in future ICPCs. Note that in October 2011, about hundreds of Live Archive problems (including the ones listed in the second edition of this book) are mirrored in the UVa Online Judge.

Figure 1.2: Left: University of Valladolid Online Judge; Right: ACM ICPC Live Archive.

The USA Computing Olympiad has a very useful training website [48] with online contests to help you learn programming and problem solving skills. This is geared for IOI participants more than for ICPC participants. Go straight to their website and train.

The Sphere Online Judge [61] is another online judge where qualified users can add their problems. This online judge is quite popular in countries like Poland, Brazil, and Vietnam. We have used this SPOJ to publish some of our self-authored problems.

Figure 1.3: Left: USACO Training Gateway; Right: Sphere Online Judge

TopCoder arranges frequent 'Single Round Match' (SRM) [32] that consists of a few problems to be solved in 1-2 hours. After the contest, you are given the chance to 'challenge' other contestants code by supplying tricky test cases. This online judge uses a rating system (red, yellow, blue, etc coders) to reward contestants who are really good at problem solving with a higher rating as opposed to more diligent contestants who happen to solve a higher number of easier problems.

1.2.7 Tip 7: Team Work (for ICPC)

This last tip is not something that is easy to teach, but here are some ideas that may be worth trying for improving your team's performance:

- Practice coding on blank paper. (This is useful when your teammate is using the computer. When it is your turn to use the computer, you can then just type the code as fast as possible rather than spending time thinking in front of the computer.)

- The 'submit and print' strategy: If your code gets an AC verdict, ignore the printout. If it still is not AC, debug your code using that printout (and let your teammate uses the computer for other problem). Beware: Debugging without the computer is not an easy skill to master.

- If your teammate is currently coding his algorithm, prepare challenges for his code by preparing hard corner-case test data (hopefully his code passes all those).

- The X-factor: Befriend your teammates *outside* of training sessions and contests.

1.3 Getting Started: The Easy Problems

Note: You can skip this section if you are a veteran participant of programming contests. This section is meant for readers who are new with competitive programming.

1.3.1 Anatomy of a Programming Contest Problem

A programming contest problem *usually* contains the following components:

- **Background story/problem description**. Usually, the easier problems are written to *deceive* contestants and made to appear difficult, for example by adding 'extra information' to create a diversion. Contestants should be able to *filter out* these

unimportant details and focus on the essential ones. For example, the entire opening paragraphs except the last sentence in UVa 579 - ClockHands are about the history of the clock and is completely unrelated to the actual problem. However, harder problems are usually written as succinctly as possible—they are already difficult enough without additional embellishment.

- **Input and Output description**. In this section, you will be given details on how the input is formatted and on how you should format your output. This part is usually written in a formal manner. A good problem should have clear input constraints as the same problem might be solvable with different algorithms for different input constraints (see Table 1.4).

- **Sample Input and Sample Output**. Problem authors usually only provide trivial test cases to contestants. The sample input/output is intended for contestants to check their basic understanding of the problem and to verify if their code can parse the given input using the given input format and produce the correct output using the given output format. Do not submit your code to the judge if it does not even pass the given sample input/output. See Section 1.2.5 about testing your code before submission.

- **Hints or Footnotes**. In some cases, the problem authors may drop hints or add footnotes to further describe the problem.

1.3.2 Typical Input/Output Routines

Multiple Test Cases

In a programming contest problem, the correctness of your code is usually determined by running your code against *several* test cases. Rather than using many individual test case files, modern programming contest problems usually use *one* test case file with multiple test cases included. In this section, we use a very simple problem as an example of a multiple-test-cases problem: Given two integers in one line, output their sum in one line. We will illustrate three possible input/output formats:

- The number of test cases is given in the first line of the input.

- The multiple test cases are terminated by special values (usually zeroes).

- The multiple test cases are terminated by the EOF (end-of-file) signal.

```
C/C++ Source Code                             | Sample Input | Sample Output
----------------------------------------------------------------------------
int TC, a, b;                                 | 3            | 3
scanf("%d", &TC); // number of test cases     | 1 2          | 12
while (TC--) { // shortcut to repeat until 0  | 5 7          | 9
  scanf("%d %d", &a, &b); // compute answer   | 6 3          |--------------
  printf("%d\n", a + b); // on the fly        |--------------|
}                                             |              |
----------------------------------------------------------------------------
int a, b;                                     | 1 2          | 3
// stop when both integers are 0              | 5 7          | 12
while (scanf("%d %d", &a, &b), (a || b))      | 6 3          | 9
  printf("%d\n", a + b);                      | 0 0          |--------------
```

```
--------------------------------------------------------------------
int a, b;                               | 1 2            | 3
// scanf returns the number of items read | 5 7          | 12
while (scanf("%d %d", &a, &b) == 2)      | 6 3           | 9
// or you can check for EOF, i.e.        |--------------|--------------
// while (scanf("%d %d", &a, &b) != EOF)  |              |
  printf("%d\n", a + b);                 |              |
```

Case Numbers and Blank Lines

Some problems with multiple test cases require the output of each test case to be numbered sequentially. Some also require a blank line *after* each test case. Let's modify the simple problem above to include the case number in the output (starting from one) with this output format: "`Case [NUMBER]: [ANSWER]`" followed by a blank line for each test case. Assuming that the input is terminated by the EOF signal, we can use the following code:

```
C/C++ Source Code                       | Sample Input | Sample Output
--------------------------------------------------------------------
int a, b, c = 1;                        | 1 2          | Case 1: 3
while (scanf("%d %d", &a, &b) != EOF)    | 5 7          |
  // notice the two '\n'                 | 6 3          | Case 2: 12
  printf("Case %d: %d\n\n", c++, a + b); |--------------|
                                         |              | Case 3: 9
                                         |              |
                                         |              |--------------
```

Some other problems require us to output blank lines only *between* test cases. If we use the approach above, we will end up with an extra new line at the end of our output, producing unnecessary 'Presentation Error' (PE) verdict. We should use the following code instead:

```
C/C++ Source Code                       | Sample Input | Sample Output
--------------------------------------------------------------------
int a, b, c = 1;                        | 1 2          | Case 1: 3
while (scanf("%d %d", &a, &b) != EOF) {  | 5 7          |
  if (c > 1) printf("\n"); // 2nd/more cases | 6 3      | Case 2: 12
  printf("Case %d: %d\n", c++, a + b);   |--------------|
}                                        |              | Case 3: 9
                                         |              |--------------
```

Variable Number of Inputs

Let's change the simple problem above slightly. For each test case (each input line), we are now given an integer k ($k \geq 1$), followed by k integers. Our task is now to output the sum of these k integers. Assuming that the input is terminated by the EOF signal and we do not require case numbering, we can use the following code:

```
C/C++ Source Code                       | Sample Input | Sample Output
--------------------------------------------------------------------
```

```
int k, ans, v;                            | 1 1          | 1
while (scanf("%d", &k) != EOF) {          | 2 3 4        | 7
  ans = 0;                                | 3 8 1 1      | 10
  while (k--) { scanf("%d", &v); ans += v; } | 4 7 2 9 3    | 21
  printf("%d\n", ans);                    | 5 1 1 1 1 1  | 5
}                                         |-------------|--------------
```

Exercise 1.3.1*: What if the problem author decides to make the input *a little more* problematic? Instead of an integer k at the beginning of each test case, you are now required to sum all integers in each test case (each line). Hint: See Section 6.2.

Exercise 1.3.2*: Rewrite all C/C++ source code in this Section 1.3.2 in Java!

1.3.3   Time to Start the Journey

There is no better way to begin your journey in competitive programming than to solve a few programming problems. To help you pick problems to start with among the ≈ 4097 problems in UVa online judge [47], we have listed some of the easiest Ad Hoc problems below. More details about Ad Hoc problems will be presented in the next Section 1.4.

- **Super Easy**
  You should get these problems AC[9] in under 7 minutes[10] each! If you are new to competitive programming, we strongly recommend that you start your journey by solving some problems from this category after completing the previous Section 1.3.2. Note: Since each category contains numerous problems for you to try, we have *highlighted* a maximum of three (3) **must try *** problems in each category. These are the problems that, we think, are more interesting or are of higher quality.

- **Easy**
  We have broken up the 'Easy' category into two smaller ones. The problems in this category are still easy, but just 'a bit' harder than the 'Super Easy' ones.

- **Medium: One Notch Above Easy**
  Here, we list some other Ad Hoc problems that may be slightly trickier (or longer) than those in the 'Easy' category.

- Super Easy Problems in the UVa Online Judge (solvable in under 7 minutes)

  1. UVa 00272 - TEX Quotes (replace all double quotes to TEX() style quotes)
  2. *UVa 01124 - Celebrity Jeopardy* (LA 2681, just echo/re-print the input again)
  3. UVa 10550 - Combination Lock (simple, do as asked)
  4. UVa 11044 - Searching for Nessy (one liner code/formula exists)
  5. **UVa 11172 - Relational Operators *** (ad hoc, very easy, one liner)
  6. UVa 11364 - Parking (linear scan to get l & r, answer $= 2 * (r - l)$)
  7. **UVa 11498 - Division of Nlogonia *** (just use if-else statements)

---

[9]Do not feel bad if you are unable to do so. There can be many reasons why a code may not get AC.
[10]Seven minutes is just a rough estimate. Some of these problems can be solved with one-liners.

8. UVa 11547 - Automatic Answer (a one liner $O(1)$ solution exists)
9. **UVa 11727 - Cost Cutting *** (sort the 3 numbers and get the median)
10. UVa 12250 - Language Detection (LA 4995, KualaLumpur10, if-else check)
11. *UVa 12279 - Emoogle Balance* (simple linear scan)
12. *UVa 12289 - One-Two-Three* (just use if-else statements)
13. *UVa 12372 - Packing for Holiday* (just check if all $L, W, H \leq 20$)
14. *UVa 12403 - Save Setu* (straightforward)
15. *UVa 12577 - Hajj-e-Akbar* (straightforward)

- Easy (just 'a bit' harder than the 'Super Easy' ones)

1. UVa 00621 - Secret Research (case analysis for only 4 possible outputs)
2. **UVa 10114 - Loansome Car Buyer *** (just simulate the process)
3. UVa 10300 - Ecological Premium (ignore the number of animals)
4. UVa 10963 - The Swallowing Ground (for two blocks to be mergable, the gaps between their columns must be the same)
5. UVa 11332 - Summing Digits (simple recursions)
6. **UVa 11559 - Event Planning *** (one linear pass)
7. UVa 11679 - Sub-prime (check if after simulation all banks have ≥ 0 reserve)
8. UVa 11764 - Jumping Mario (one linear scan to count high+low jumps)
9. **UVa 11799 - Horror Dash *** (one linear scan to find the max value)
10. UVa 11942 - Lumberjack Sequencing (check if input is sorted asc/descending)
11. UVa 12015 - Google is Feeling Lucky (traverse the list twice)
12. *UVa 12157 - Tariff Plan* (LA 4405, KualaLumpur08, compute and compare)
13. *UVa 12468 - Zapping* (easy; there are only 4 possibilities)
14. *UVa 12503 - Robot Instructions* (easy simulation)
15. *UVa 12554 - A Special ... Song* (simulation)
16. IOI 2010 - Cluedo (use 3 pointers)
17. IOI 2010 - Memory (use 2 linear pass)

- Medium: One Notch Above Easy (may take 15-30 minutes, but not too hard)

1. UVa 00119 - Greedy Gift Givers (simulate give and receive process)
2. **UVa 00573 - The Snail *** (simulation, beware of boundary cases!)
3. UVa 00661 - Blowing Fuses (simulation)
4. **UVa 10141 - Request for Proposal *** (solvable with one linear scan)
5. UVa 10324 - Zeros and Ones (simplify using 1D array: change counter)
6. UVa 10424 - Love Calculator (just do as asked)
7. UVa 10919 - Prerequisites? (process the requirements as the input is read)
8. **UVa 11507 - Bender B. Rodriguez ... *** (simulation, if-else)
9. UVa 11586 - Train Tracks (TLE if brute force, find the pattern)
10. UVa 11661 - Burger Time? (linear scan)
11. *UVa 11683 - Laser Sculpture* (one linear pass is enough)
12. UVa 11687 - Digits (simulation; straightforward)
13. UVa 11956 - Brain**** (simulation; ignore '.')
14. *UVa 12478 - Hardest Problem ...* (try one of the eight names)
15. IOI 2009 - Garage (simulation)
16. IOI 2009 - POI (sort)

1.4 The Ad Hoc Problems

We will terminate this chapter by discussing the first proper problem type in the ICPCs and IOIs: The Ad Hoc problems. According to USACO [48], the Ad Hoc problems are problems that 'cannot be classified anywhere else' since each problem description and its corresponding solution are 'unique'. Many Ad Hoc problems are easy (as shown in Section 1.3), but this does not apply to all Ad Hoc problems.

Ad Hoc problems frequently appear in programming contests. In ICPC, $\approx$ 1-2 problems out of every $\approx$ 10 problems are Ad Hoc problems. If the Ad Hoc problem is easy, it will usually be the first problem solved by the teams in a programming contest. However, there were cases where solutions to the Ad Hoc problems were too complicated to implement, causing some teams to strategically defer them to the last hour. In an ICPC regional contest with about 60 teams, your team would rank in the lower half (rank 30-60) if you can *only* solve Ad Hoc problems.

In IOI 2009 and 2010, there has been 1 easy task per competition day[11], usually an (Easy) Ad Hoc task. If you are an IOI contestant, you will definitely not win any medals for just solving the 2 easy Ad Hoc tasks over the 2 competition days. However, the faster you can clear these 2 easy tasks, the more time that you will have to work on the other $2 \times 3 = 6$ challenging tasks.

We have listed **many** Ad Hoc problems that we have solved in the UVa Online Judge [47] in the several categories below. We believe that you can solve most of these problems *without* using the advanced data structures or algorithms that will be discussed in the later chapters. Many of these Ad Hoc problems are 'simple' but some of them maybe 'tricky'. Try to solve a few problems from each category before reading the next chapter.

Note: A small number of problems, although listed as part of Chapter 1, may require knowledge from subsequent chapters, e.g. knowledge of linear data structures (arrays) in Section 2.2, knowledge of backtracking in Section 3.2, etc. You can revisit these harder Ad Hoc problems after you have understood the required concepts.

The categories:

- **Game (Card)**
  There are lots of Ad Hoc problems involving popular games. Many are related to card games. You will usually need to parse the input strings (see Section 6.3) as playing cards have both suits (D/Diamond/$\diamond$, C/Club/$\clubsuit$, H/Heart/$\heartsuit$, and S/Spades/$\spadesuit$) and ranks (usually: $2 < 3 < \ldots < 9 < $ T/Ten $<$ J/Jack $<$ Q/Queen $<$ K/King $<$ A/Ace[12]). It may be a good idea to map these troublesome strings to integer indices. For example, one possible mapping is to map D2 $\rightarrow$ 0, D3 $\rightarrow$ 1, ..., DA $\rightarrow$ 12, C2 $\rightarrow$ 13, C3 $\rightarrow$ 14, ..., SA $\rightarrow$ 51. Then, you can work with the integer indices instead.

- **Game (Chess)**
  Chess is another popular game that sometimes appears in programming contest problems. Some of these problems are Ad Hoc and listed in this section. Some of them are combinatorial with tasks like counting how many ways there are to place 8-queens in 8×8 chess board. These are listed in Chapter 3.

- **Game (Others)**, easier and harder (or more tedious)
  Other than card and chess games, many other popular games have made their way into programming contests: Tic Tac Toe, Rock-Paper-Scissors, Snakes/Ladders, BINGO,

---

[11]This was no longer true in IOI 2011-2012 as the easier scores are inside subtask 1 of each task.
[12]In some other arrangements, A/Ace < 2.

Bowling, etc. Knowing the details of these games may be helpful, but most of the game rules are given in the problem description to avoid disadvantaging contestants who are unfamiliar with the games.

- Problems related to **Palindromes**
  These are also classic problems. A palindrome is a word (or a sequence) that can be read the same way in either direction. The most common strategy to check if a word is palindromic is to loop from the first character to the *middle* one and check if the characters match in the corresponding position from the back. For example, '**A**B<u>C</u>D<u>C</u>B**A**' is a palindrome. For some harder palindrome-related problems, you may want to check Section 6.5 for Dynamic Programming solutions.

- Problems related to **Anagrams**
  This is yet another class of classic problems. An anagram is a word (or phrase) whose letters can be rearranged to obtain another word (or phrase). The common strategy to check if two words are anagrams is to sort the letters of the words and compare the results. For example, take `wordA = 'cab'`, `wordB = 'bca'`. After sorting, `wordA = 'abc'` and `wordB = 'abc'` too, so they are anagrams. See Section 2.2 for various sorting techniques.

- Interesting **Real Life** Problems, easier and harder (or more tedious)
  This is one of the most interesting problem categories in the UVa Online Judge. We believe that real life problems like these are interesting to those who are new to Computer Science. The fact that we write programs to solve real life problems can be an additional motivational boost. Who knows, you might stand to gain new (and interesting) information from the problem description!

- Ad Hoc problems involving **Time**
  These problems utilize time concepts such as dates, times, and calendars. These are also real life problems. As mentioned earlier, these problems can be a little more interesting to solve. Some of these problems will be far easier to solve if you have mastered the Java GregorianCalendar class as it has many library functions that deal with time.

- **'Time Waster'** problems
  These are Ad Hoc problems that are written specifically to make the required solution long and tedious. These problems, if they do appear in a programming contest, would determine the team with the most *efficient* coder—someone who can implement complicated but still accurate solutions under time constraints. Coaches should consider adding such problems in their training programmes.

- Ad Hoc problems in **other chapters**
  There are many other Ad Hoc problems which we have shifted to other chapters since they required knowledge above basic programming skills.

  - Ad Hoc problems involving the usage of basic linear data structures (especially arrays) are listed in Section 2.2.
  - Ad Hoc problems involving mathematical computation are listed in Section 5.2.
  - Ad Hoc problems involving string processing are listed in Section 6.3.
  - Ad Hoc problems involving basic geometry are listed in Section 7.2.
  - Ad Hoc problems listed in Chapter 9.

Tips: After solving a number of programming problems, you begin to realize a pattern in your solutions. Certain idioms are used frequently enough in competitive programming implementation for shortcuts to be useful. From a C/C++ perspective, these idioms might include: Libraries to be included (cstdio, cmath, cstring, etc), data type shortcuts (ii, vii, vi, etc), basic I/O routines (freopen, multiple input format, etc), loop macros (e.g. `#define REP(i, a, b) for (int i = int(a); i <= int(b); i++)`, etc), and a few others. A competitive programmer using C/C++ can store these in a header file like 'competitive.h'. With such a header, the solution to every problem begins with a simple `#include<competitive.h>`. However, this tips should not be used beyond competitive programming, especially in software industry.

Programming Exercises related to Ad Hoc problems:

- Game (Card)

    1. UVa 00162 - Beggar My Neighbour (card game simulation; straightforward)
    2. **UVa 00462 - Bridge Hand Evaluator *** (simulation; card)
    3. UVa 00555 - Bridge Hands (card game)
    4. UVa 10205 - Stack 'em Up (card game)
    5. *UVa 10315 - Poker Hands* (tedious problem)
    6. **UVa 10646 - What is the Card? *** (shuffle cards with some rule and then get certain card)
    7. UVa 11225 - Tarot scores (another card game)
    8. UVa 11678 - Card's Exchange (actually just an array manipulation problem)
    9. ***UVa 12247 - Jollo **** (interesting card game; simple, but requires good logic to get all test cases correct)

- Game (Chess)

    1. UVa 00255 - Correct Move (check the validity of chess moves)
    2. **UVa 00278 - Chess *** (ad hoc, chess, closed form formula exists)
    3. **UVa 00696 - How Many Knights *** (ad hoc, chess)
    4. UVa 10196 - Check The Check (ad hoc chess game, tedious)
    5. **UVa 10284 - Chessboard in FEN *** (FEN = Forsyth-Edwards Notation is a standard notation for describing board positions in a chess game)
    6. UVa 10849 - Move the bishop (chess)
    7. UVa 11494 - Queen (ad hoc, chess)

- Game (Others), Easier

    1. UVa 00340 - Master-Mind Hints (determine strong and weak matches)
    2. **UVa 00489 - Hangman Judge *** (just do as asked)
    3. *UVa 00947 - Master Mind Helper* (similar to UVa 340)
    4. **UVa 10189 - Minesweeper *** (simulate Minesweeper, similar to UVa 10279)
    5. UVa 10279 - Mine Sweeper (a 2D array helps, similar to UVa 10189)
    6. UVa 10409 - Die Game (just simulate the die movement)
    7. UVa 10530 - Guessing Game (use a 1D flag array)
    8. **UVa 11459 - Snakes and Ladders *** (simulate it, similar to UVa 647)
    9. *UVa 12239 - Bingo* (try all 90^2 pairs, see if all numbers in `[0..N]` are there)

- Game (Others), Harder (more tedious)
    1. UVa 00114 - Simulation Wizardry (simulation of pinball machine)
    2. UVa 00141 - The Spot Game (simulation, pattern check)
    3. UVa 00220 - Othello (follow the game rules, a bit tedious)
    4. UVa 00227 - Puzzle (parse the input, array manipulation)
    5. UVa 00232 - Crossword Answers (complex array manipulation problem)
    6. UVa 00339 - SameGame Simulation (follow problem description)
    7. UVa 00379 - HI-Q (follow problem description)
    8. **UVa 00584 - Bowling *** (simulation, games, reading comprehension)
    9. UVa 00647 - Chutes and Ladders (childhood board game, also see UVa 11459)
    10. UVa 10363 - Tic Tac Toe (check validity of Tic Tac Toe game, tricky)
    11. **UVa 10443 - Rock, Scissors, Paper *** (2D arrays manipulation)
    12. **UVa 10813 - Traditional BINGO *** (follow the problem description)
    13. UVa 10903 - Rock-Paper-Scissors ... (count win+losses, output win average)
- Palindrome
    1. UVa 00353 - Pesky Palindromes (brute force all substring)
    2. **UVa 00401 - Palindromes *** (simple palindrome check)
    3. UVa 10018 - Reverse and Add (ad hoc, math, palindrome check)
    4. **UVa 10945 - Mother Bear *** (palindrome)
    5. **UVa 11221 - Magic Square Palindrome *** (we deal with a matrix)
    6. UVa 11309 - Counting Chaos (palindrome check)
- Anagram
    1. UVa 00148 - Anagram Checker (uses backtracking)
    2. **UVa 00156 - Ananagram *** (easier with `algorithm::sort`)
    3. **UVa 00195 - Anagram *** (easier with `algorithm::next_permutation`)
    4. **UVa 00454 - Anagrams *** (anagram)
    5. UVa 00630 - Anagrams (II) (ad hoc, string)
    6. UVa 00642 - Word Amalgamation (go through the given small dictionary for the list of possible anagrams)
    7. UVa 10098 - Generating Fast, Sorted ... (very similar to UVa 195)
- Interesting Real Life Problems, Easier
    1. **UVa 00161 - Traffic Lights *** (this is a typical situation on the road)
    2. UVa 00187 - Transaction Processing (an accounting problem)
    3. UVa 00362 - 18,000 Seconds Remaining (typical file download situation)
    4. **UVa 00637 - Booklet Printing *** (application in printer driver software)
    5. *UVa 00857 - Quantiser* (MIDI, application in computer music)
    6. UVa 10082 - WERTYU (this typographical error happens to us sometimes)
    7. UVa 10191 - Longest Nap (you may want to apply this to your own schedule)
    8. UVa 10528 - Major Scales (music knowledge is in the problem description)
    9. UVa 10554 - Calories from Fat (are you concerned with your weights?)
    10. **UVa 10812 - Beat the Spread *** (be careful with boundary cases!)
    11. UVa 11530 - SMS Typing (handphone users encounter this problem everyday)
    12. *UVa 11945 - Financial Management* (a bit output formatting)
    13. UVa 11984 - A Change in Thermal Unit (F° to C° conversion and vice versa)
    14. *UVa 12195 - Jingle Composing* (count the number of correct measures)
    15. *UVa 12555 - Baby Me* (one of the first question asked when a new baby is born; requires a bit of input processing)

- Interesting Real Life Problems, Harder (more tedious)

  1. UVa 00139 - Telephone Tangles (calculate phone bill; string manipulation)
  2. UVa 00145 - Gondwanaland Telecom (similar nature with UVa 139)
  3. *UVa 00333 - Recognizing Good ISBNs* (note: this problem has 'buggy' test data with blank lines that potentially cause lots of 'Presentation Errors')
  4. UVa 00346 - Getting Chorded (musical chord, major/minor)
  5. **UVa 00403 - Postscript *** (emulation of printer driver, tedious)
  6. *UVa 00447 - Population Explosion* (life simulation model)
  7. UVa 00448 - OOPS (tedious 'hexadecimal' to 'assembly language' conversion)
  8. *UVa 00449 - Majoring in Scales* (easier if you have a musical background)
  9. UVa 00457 - Linear Cellular Automata (simplified 'game of life' simulation; similar idea with UVa 447; explore the Internet for that term)
  10. UVa 00538 - Balancing Bank Accounts (the problem's premise is quite true)
  11. **UVa 00608 - Counterfeit Dollar *** (classical problem)
  12. UVa 00706 - LC-Display (what we see in old digital display)
  13. **UVa 01061 - Consanguine Calculations *** (LA 3736 - WorldFinals Tokyo07, consanguine = blood; this problem asks possible combinations of blood types and Rh factor; solvable by trying all eight possible blood + Rh types with the information given in the problem description)
  14. UVa 10415 - Eb Alto Saxophone Player (about musical instruments)
  15. UVa 10659 - Fitting Text into Slides (typical presentation programs do this)
  16. UVa 11223 - O: dah, dah, dah (tedious morse code conversion)
  17. UVa 11743 - Credit Check (Luhn's algorithm to check credit card numbers; search the Internet to learn more)
  18. *UVa 12342 - Tax Calculator* (tax computation can be tricky indeed)

- Time

  1. UVa 00170 - Clock Patience (simulation, time)
  2. UVa 00300 - Maya Calendar (ad hoc, time)
  3. **UVa 00579 - Clock Hands *** (ad hoc, time)
  4. **UVa 00893 - Y3K *** (use Java GregorianCalendar; similar to UVa 11356)
  5. UVa 10070 - Leap Year or Not Leap ... (more than ordinary leap years)
  6. *UVa 10339 - Watching Watches* (find the formula)
  7. UVa 10371 - Time Zones (follow the problem description)
  8. UVa 10683 - The decadary watch (simple clock system conversion)
  9. UVa 11219 - How old are you? (be careful with boundary cases!)
  10. UVa 11356 - Dates (very easy if you use Java GregorianCalendar)
  11. UVa 11650 - Mirror Clock (some mathematics required)
  12. UVa 11677 - Alarm Clock (similar idea with UVa 11650)
  13. **UVa 11947 - Cancer or Scorpio *** (easier with Java GregorianCalendar)
  14. UVa 11958 - Coming Home (be careful with 'past midnight')
  15. UVa 12019 - Doom's Day Algorithm (Gregorian Calendar; get DAY_OF_WEEK)
  16. UVa 12136 - Schedule of a Married Man (LA 4202, Dhaka08, check time)
  17. *UVa 12148 - Electricity* (easy with Gregorian Calendar; use method 'add' to add one day to previous date and see if it is the same as the current date)
  18. *UVa 12439 - February 29* (inclusion-exclusion; lots of corner cases; be careful)
  19. *UVa 12531 - Hours and Minutes* (angles between two clock hands)

- 'Time Waster' Problems

  1. UVa 00144 - Student Grants (simulation)
  2. *UVa 00214 - Code Generation* (just simulate the process; be careful with subtract (-), divide (/), and negate (@), tedious)
  3. UVa 00335 - Processing MX Records (simulation)
  4. UVa 00337 - Interpreting Control ... (simulation, output related)
  5. UVa 00349 - Transferable Voting (II) (simulation)
  6. UVa 00381 - Making the Grade (simulation)
  7. UVa 00405 - Message Routing (simulation)
  8. **UVa 00556 - Amazing *** (simulation)
  9. *UVa 00603 - Parking Lot* (simulate the required process)
  10. *UVa 00830 - Shark* (very hard to get AC, one minor error = WA)
  11. *UVa 00945 - Loading a Cargo Ship* (simulate the given cargo loading process)
  12. UVa 10033 - Interpreter (adhoc, simulation)
  13. *UVa 10134 - AutoFish* (must be very careful with details)
  14. UVa 10142 - Australian Voting (simulation)
  15. UVa 10188 - Automated Judge Script (simulation)
  16. UVa 10267 - Graphical Editor (simulation)
  17. *UVa 10961 - Chasing After Don Giovanni* (tedious simulation)
  18. UVa 11140 - Little Ali's Little Brother (ad hoc)
  19. UVa 11717 - Energy Saving Micro... (tricky simulation)
  20. **UVa 12060 - All Integer Average *** (LA 3012, Dhaka04, output format)
  21. **UVa 12085 - Mobile Casanova *** (LA 2189, Dhaka06, watch out for PE)
  22. *UVa 12608 - Garbage Collection* (simulation with several corner cases)

1.5   Solutions to Non-Starred Exercises

Exercise 1.1.1: A simple test case to break greedy algorithms is $N = 2$, $\{(2, 0), (2, 1), (0, 0),$ $(4, 0)\}$. A greedy algorithm will incorrectly pair $\{(2, 0), (2, 1)\}$ and $\{(0, 0), (4, 0)\}$ with a 5.000 cost while the optimal solution is to pair $\{(0, 0), (2, 0)\}$ and $\{(2, 1), (4, 0)\}$ with cost 4.236.

Exercise 1.1.2: For a Naïve Complete Search like the one outlined in the body text, one needs up to $_{16}C_2 \times_{14} C_2 \times \ldots \times_2 C_2$ for the largest test case with $N = 8$—far too large. However, there are ways to prune the search space so that Complete Search can still work. For an extra challenge, attempt **Exercise 1.1.3*!**

Exercise 1.2.1: The complete Table 1.3 is shown below.

UVa	Title	Problem Type	Hint
10360	Rat Attack	Complete Search or DP	Section 3.2
10341	Solve It	Divide & Conquer (Bisection Method)	Section 3.3
11292	Dragon of Loowater	Greedy (Non Classical)	Section 3.4
11450	Wedding Shopping	DP (Non Classical)	Section 3.5
10911	Forming Quiz Teams	DP with bitmasks (Non Classical)	Section 8.3.1
11635	Hotel Booking	Graph (Decomposition: Dijkstra's + BFS)	Section 8.4
11506	Angry Programmer	Graph (Min Cut/Max Flow)	Section 4.6
10243	Fire! Fire!! Fire!!!	DP on Tree (Min Vertex Cover)	Section 4.7.1
10717	Mint	Decomposition: Complete Search + Math	Section 8.4
11512	GATTACA	String (Suffix Array, LCP, LRS)	Section 6.6
10065	Useless Tile Packers	Geometry (Convex Hull + Area of Polygon)	Section 7.3.7

Exercise 1.2.2: The answers are:

1. (b) Use a priority queue data structure (heap) (Section 2.3).

2. (b) Use 2D Range Sum Query (Section 3.5.2).

3. If list L is static, (b) Simple Array that is pre-processed with Dynamic Programming (Section 2.2 & 3.5). If list L is dynamic, then (g) Fenwick Tree is a better answer (easier to implement than (f) Segment Tree).

4. (a) Yes, a complete search is possible (Section 3.2).

5. (a) $O(V + E)$ Dynamic Programming (Section 3.5, 4.2.5, & 4.7.1).
   However, (c) $O((V + E) \log V)$ Dijkstra's algorithm is also possible since the extra $O(\log V)$ factor is still 'small' for V up to $100K$.

6. (a) Sieve of Eratosthenes (Section 5.5.1).

7. (b) The naïve approach above will not work. We must (prime) factorize $n!$ and m and see if the (prime) factors of m can be found in the factors of $n!$ (Section 5.5.5).

8. (b) No, we must find another way. First, find the Convex Hull of the N points in $O(n \log n)$ (Section 7.3.7). Let the number of points in $CH(S) = k$. As the points are randomly scattered, k will be much smaller than N. Then, find the two farthest points by examining all pairs of points in the $CH(S)$ in $O(k^2)$.

9. (b) The naïve approach is too slow. Use KMP or Suffix Array (Section 6.4 or 6.6)!

Exercise 1.2.3: The Java code is shown below:

```java
// Java code for task 1, assuming all necessary imports have been done
class Main {
  public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    double d = sc.nextDouble();
    System.out.printf("%7.3f\n", d);          // yes, Java has printf too!
} }
```

```cpp
// C++ code for task 2, assuming all necessary includes have been done
int main() {
  double pi = 2 * acos(0.0);    // this is a more accurate way to compute pi
  int n; scanf("%d", &n);
  printf("%.*lf\n", n, pi);     // this is the way to manipulate field width
}
```

```java
// Java code for task 3, assuming all necessary imports have been done
class Main {
  public static void main(String[] args) {
    String[] names = new String[]
      { "", "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat" };
    Calendar calendar = new GregorianCalendar(2010, 7, 9); // 9 August 2010
        // note that month starts from 0, so we need to put 7 instead of 8
    System.out.println(names[calendar.get(Calendar.DAY_OF_WEEK)]); // "Wed"
} }
```

```cpp
// C++ code for task 4, assuming all necessary includes have been done
#define ALL(x) x.begin(), x.end()
#define UNIQUE(c) (c).resize(unique(ALL(c)) - (c).begin())

int main() {
  int a[] = {1, 2, 2, 2, 3, 3, 2, 2, 1};
  vector<int> v(a, a + 9);
  sort(ALL(v)); UNIQUE(v);
  for (int i = 0; i < (int)v.size(); i++) printf("%d\n", v[i]);
}
```

```cpp
// C++ code for task 5, assuming all necessary includes have been done
typedef pair<int, int> ii;       // we will utilize the natural sort order
typedef pair<int, ii> iii;    // of the primitive data types that we paired

int main() {
  iii A = make_pair(ii(5, 24), -1982);                // reorder DD/MM/YYYY
  iii B = make_pair(ii(5, 24), -1980);                        // to MM, DD,
  iii C = make_pair(ii(11, 13), -1983);       // and then use NEGATIVE YYYY
  vector<iii> birthdays;
  birthdays.push_back(A); birthdays.push_back(B); birthdays.push_back(C);
  sort(birthdays.begin(), birthdays.end());              // that's all :)
}
```

```cpp
// C++ code for task 6, assuming all necessary includes have been done
int main() {
  int n = 5, L[] = {10, 7, 5, 20, 8}, v = 7;
  sort(L, L + n);
  printf("%d\n", binary_search(L, L + n, v));
}

// C++ code for task 7, assuming all necessary includes have been done
int main() {
  int p[10], N = 10; for (int i = 0; i < N; i++) p[i] = i;
  do {
    for (int i = 0; i < N; i++) printf("%c ", 'A' + p[i]);
    printf("\n");
  }
  while (next_permutation(p, p + N));
}

// C++ code for task 8, assuming all necessary includes have been done
int main() {
  int p[20], N = 20;
  for (int i = 0; i < N; i++) p[i] = i;
  for (int i = 0; i < (1 << N); i++) {
    for (int j = 0; j < N; j++)
      if (i & (1 << j))                                // if bit j is on
        printf("%d ", p[j]);                           // this is part of set
    printf("\n");
  } }
```

```java
// Java code for task 9, assuming all necessary imports have been done
class Main {
  public static void main(String[] args) {
    String str = "FF"; int X = 16, Y = 10;
    System.out.println(new BigInteger(str, X).toString(Y));
} }

// Java code for task 10, assuming all necessary imports have been done
class Main {
  public static void main(String[] args) {
    String S = "line: a70 and z72 will be replaced, aa24 and a872 will not";
    System.out.println(S.replaceAll("(^| )+[a-z][0-9][0-9]( |$)+", " *** "));
} }

// Java code for task 11, assuming all necessary imports have been done
class Main {
  public static void main(String[] args) throws Exception {
    ScriptEngineManager mgr = new ScriptEngineManager();
    ScriptEngine engine = mgr.getEngineByName("JavaScript");      // "cheat"
    Scanner sc = new Scanner(System.in);
    while (sc.hasNextLine()) System.out.println(engine.eval(sc.nextLine()));
} }
```

Exercise 1.2.4: Situational considerations are in brackets:

1. You receive a WA verdict for a very easy problem. What should you do?

   (a) Abandon this problem for another. (**Not ok, your team will lose out.**)
   (b) Improve the performance of your solution. (**Not useful.**)
   (c) Create tricky test cases to find the bug. (**The most logical answer.**)
   (d) (In team contests): Ask your team mate to re-do the problem. (**This could be feasible as you might have had some wrong assumptions about the problem. Thus, you should refrain from telling the details about the problem to your team mate who will re-do the problem. Still, your team will lose precious time.**)

2. You receive a TLE verdict for your $O(N^3)$ solution.
   However, the maximum N is just 100. What should you do?

   (a) Abandon this problem for another. (**Not ok, your team will lose out.**)
   (b) Improve the performance of your solution. (**Not ok, we should not get TLE with an $O(N^3)$ algorithm if $N \leq 400$.**)
   (c) Create tricky test cases to find the bug. (**This is the answer—maybe your program runs into an accidental infinite loop in some test cases.**)

3. Follow up to Question 2: What if the maximum N is 100.000?
   (**If $N > 400$, you may have no choice but to improve the performance of the current algorithm or use a another faster algorithm.**)

4. Another follow up to Question 2: What if the maximum N is 1.000, the output only depends on the size of input N, and you still have *four hours* of competition time left?
   (**If the output only depends on N, you *may* be able to pre-calculate all possible solutions by running your $O(N^3)$ algorithm in the background, letting your team mate use the computer first. Once your $O(N^3)$ solution terminates, you have all the answers. Submit the $O(1)$ answer instead if it does not exceed 'source code size limit' imposed by the judge.**)

5. You receive an RTE verdict. Your code (seems to) execute perfectly on your machine. What should you do?
   (**The most common causes of RTEs are usually array sizes that are too small or stack overflow/infinite recursion errors. Design test cases that can trigger these errors in your code.**)

6. Thirty minutes into the contest, you take a glance at the leader board. There are *many* other teams that have solved a problem X that your team has not attempted. What should you do?
   (**One team member should immediately attempt problem X as it may be relatively easy. Such a situation is really a bad news for your team as it is a major set-back to getting a good rank in the contest.**)

7. Midway through the contest, you take a glance at the leader board. The leading team (assume that it is not your team) has just solved problem Y. What should you do?
   (**If your team is not the 'pace-setter', then it is a good idea to 'ignore' what the leading team is doing and concentrate instead on solving the problems that your team has identified to be 'solvable'. By mid-contest your team must have read all the problems in the problem set and roughly identified the problems solvable with your team's current abilities.**)

8. Your team has spent two hours on a nasty problem. You have submitted several implementations by different team members. All submissions have been judged incorrect. You have no idea what's wrong. What should you do?
   (It is time to give up solving this problem. Do not hog the computer, let your team mate solves another problem. Either your team has really misunderstood the problem or in a very rare case, the judge solution is actually wrong. In any case, this is not a good situation for your team.)

9. There is one hour to go before the end of the contest. You have 1 WA code and 1 fresh idea for *another* problem. What should you (or your team) do?
   (In chess terminology, this is called the 'end game' situation.)

   (a) Abandon the problem with the WA code, switch to the other problem in an attempt to solve one more problem.**(OK in individual contests like IOI.)**

   (b) Insist that you have to debug the WA code. There is not enough time to start working on a new problem. **(If the idea for another problem involves complex and tedious code, then deciding to focus on the WA code may be a good idea rather than having two incomplete/'non AC' solutions.)**

   (c) (In ICPC): Print the WA code. Ask two other team members to scrutinize it while you switch to that other problem in an attempt to solve *two* more problems.
   (If the solution for the other problem can be coded in less than 30 minutes, then implement it while your team mates try to find the bug for the WA code by studying the printed copy.)

Figure 1.4: Some references that inspired the authors to write this book

1.6 Chapter Notes

This chapter, as well as subsequent chapters are supported by many textbooks (see Figure 1.4 in the previous page) and Internet resources. Here are some additional references:

- To improve your typing skill as mentioned in Tip 1, you may want to play the many typing games available online.

- Tip 2 is adapted from the introduction text in USACO training gateway [48].

- More details about Tip 3 can be found in many CS books, e.g. Chapter 1-5, 17 of [7].

- Online references for Tip 4:
  http://www.cppreference.com and http://www.sgi.com/tech/stl/ for C++ STL;
  http://docs.oracle.com/javase/7/docs/api/ for Java API.
  You do not have to memorize all library functions, but it is useful to memorize functions that you frequently use.

- For more insights on better testing (Tip 5), a slight detour to software engineering books may be worth trying.

- There are many other Online Judges apart from those mentioned in Tip 6, e.g.

  - Codeforces, http://codeforces.com/,
  - Peking University Online Judge, (POJ) http://poj.org,
  - Zhejiang University Online Judge, (ZOJ) http://acm.zju.edu.cn,
  - Tianjin University Online Judge, http://acm.tju.edu.cn/toj,
  - Ural State University (Timus) Online Judge, http://acm.timus.ru,
  - URI Online Judge, http://www.urionlinejudge.edu.br, etc.

- For a note regarding team contest (Tip 7), read [16].

In this chapter, we have introduced the world of competitive programming to you. However, a competitive programmer must be able to solve more than just Ad Hoc problems in a programming contest. We hope that you will enjoy the ride and fuel your enthusiasm by reading up on and learning new concepts in the *other* chapters of this book. Once you have finished reading the book, re-read it once more. On the second time, attempt and solve the ≈ 238 written exercises and the ≈ 1675 programming exercises.

Statistics	First Edition	Second Edition	Third Edition
Number of Pages	13	19 (+46%)	32 (+68%)
Written Exercises	4	4	6+3*=9 (+125%)
Programming Exercises	34	160 (+371%)	173 (+8%)

Chapter 2

Data Structures and Libraries

If I have seen further it is only by standing on the shoulders of giants.
— **Isaac Newton**

2.1 Overview and Motivation

A data structure (DS) is a means of storing and organizing data. Different data structures have different strengths. So when designing an algorithm, it is important to pick one that allows for efficient insertions, searches, deletions, queries, and/or updates, depending on what your algorithm needs. Although a data structure does not in itself solve a (programming contest) problem (the algorithm operating on it does), using an appropriately efficient data structure for a problem may be the difference between passing or exceeding the problem's time limit. There can be many ways to organize the same data and sometimes one way is better than the other in some contexts. We will illustrate this several times in this chapter. A keen familiarity with the data structures and libraries discussed in this chapter is critically important for understanding the algorithms that use them in subsequent chapters.

As stated in the preface of this book, we **assume** that you are *familiar* with the basic data structures listed in Section 2.2-2.3 and thus we will **not** review them in this book. Instead, we will simply highlight the fact that there exist built-in implementations for these elementary data structures in the C++ STL and Java API[1]. If you feel that you are not entirely familiar with any of the terms or data structures mentioned in Section 2.2-2.3, please review those particular terms and concepts in the various reference books[2] that cover them, including classics such as the "Introduction to Algorithms" [7], "Data Abstraction and Problem Solving" [5, 54], "Data Structures and Algorithms" [12], etc. Continue reading this book only when you understand at least the *basic concepts* behind these data structures.

Note that for competitive programming, you only need to know enough about these data structures to be able to select and to *use* the correct data structures for each given contest problem. You should understand the strengths, weaknesses, and time/space complexities of typical data structures. The theory behind them is definitely good reading, but can often be skipped or skimmed through, since the built-in libraries provide ready-to-use and highly reliable implementations of otherwise complex data structures. This is *not* a good practice, but you will find that it is often sufficient. Many (younger) contestants have been able to utilize the efficient (with a $O(\log n)$ complexity for most operations) C++ STL `map` (or

---

[1]Even in this third edition, we *still* primarily use C++ code to illustrate implementation techniques. The Java equivalents can be found in the supporting website of this book.

[2]Materials in Section 2.2-2.3 are usually covered in year one *Data Structures* CS curriculae. High school students aspiring to take part in the IOI are encouraged to engage in independent study on such material.

Java `TreeMap`) implementations to store dynamic collections of key-data pairs without an understanding that the underlying data structure is a *balanced Binary Search Tree*, or use the C++ STL `priority_queue` (or Java `PriorityQueue`) to order a queue of items without understanding that the underlying data structure is a *(usually Binary) Heap*. Both data structures are typically taught in year one Computer Science curriculae.

This chapter is divided into three parts. Section 2.2 contains basic *linear* data structures and the basic operations they support. Section 2.3 covers basic *non-linear* data structures such as (balanced) Binary Search Trees (BST), (Binary) Heaps, and Hash Tables, as well as their basic operations. The discussion of each data structure in Section 2.2-2.3 is brief, with an emphasis on the important *library routines* that exist for manipulating the data structures. However, special data structures that are common in programming contests, such as bitmask and several bit manipulation techniques (see Figure 2.1) are discussed in more detail. Section 2.4 contains *more* data structures for which there exist no built-in implementation, and thus require us to build *our own* libraries. Section 2.4 has a more in-depth discussion than Section 2.2-2.3.

Value-Added Features of this Book

As this chapter is the first that dives into the heart of competitive programming, we will now take the opportunity to highlight several value-added features of this book that you will see in this and the following chapters.

A key feature of this book is its accompanying collection of *efficient, fully-implemented examples* in both C/C++ and Java that many other Computer Science books lack, stopping at the 'pseudo-code level' in their demonstration of data structures and algorithms. This feature has been in the book since the very first edition. The important parts of the source code have been included in the book[3] and the full source code is hosted at `sites.google.com/site/stevenhalim/home/material`. The reference to each source file is indicated in the body text as a box like the one shown below.

Source code: `chx_yy_name.cpp/java`

Another strength of this book is the collection of both written and programming exercises (mostly supported by the UVa Online Judge [47] and integrated with uHunt—see Appendix A). In the *third* edition, we have added *many more* written exercises. We have classified the written exercises into *non-starred* and *starred* ones. The non-starred written exercises are meant to be used mainly for self-checking purposes; solutions are given at the back of each chapter. The starred written exercises can be used for extra challenges; we do not provide solutions for these but may instead provide some helpful hints.

In the *third* edition, we have added visualizations[4] for many data structures and algorithms covered in this book [27]. We believe that these visualizations will be a huge benefit to the visual learners in our reader base. At this point in time (24 May 2013), the visualizations are hosted at: `www.comp.nus.edu.sg/~stevenha/visualization`. The reference to each visualization is included in the body text as a box like the one shown below.

Visualization: `www.comp.nus.edu.sg/~stevenha/visualization`

---

[3]However, we have chosen not to include code from Section 2.2-2.3 in the body text because they are mostly 'trivial' for many readers, except perhaps for a few useful tricks.

[4]They are built with HTML5 canvas and JavaScript technology.

2.2 Linear DS with Built-in Libraries

A data structure is classified as a *linear* data structure if its elements form a linear sequence, i.e. its elements are arranged from left to right (or top to bottom). Mastery of these basic linear data structures below is critical in today's programming contests.

- Static Array (native support in both C/C++ and Java)
  This is clearly the most commonly used data structure in programming contests. Whenever there is a collection of sequential data to be stored and later accessed using their *indices*, the static array is the most natural data structure to use. As the maximum input size is usually mentioned in the problem statement, the array size can be declared to be the maximum input size, with a small extra buffer (sentinel) for safety—to avoid the unnecessary 'off by one' RTE. Typically, 1D, 2D, and 3D arrays are used in programming contests—problems rarely require arrays of higher dimension. Typical array operations include accessing elements by their indices, sorting elements, performing a linear scan or a binary search on a sorted array.

- Dynamically-Resizeable Array: C++ STL `vector` (Java `ArrayList` (faster) or `Vector`)
  This data structure is similar to the static array, except that it is designed to handle runtime resizing natively. It is better to use a `vector` in place of an array if the size of the sequence of elements is unknown at compile-time. Usually, we initialize the size (`reserve()` or `resize()`) with the estimated size of the collection for better performance. Typical C++ STL `vector` operations used in competitive programming include `push_back()`, `at()`, the `[]` operator, `assign()`, `clear()`, `erase()`, and `iterator`s for traversing the contents of `vector`s.

> Source code: `ch2_01_array_vector.cpp/java`

It is appropriate to discuss two operations commonly performed on Arrays: **Sorting** and **Searching**. These two operations are well supported in C++ and Java.

There are *many* sorting algorithms mentioned in CS books [7, 5, 54, 12, 40, 58], e.g.

1. $O(n^2)$ comparison-based sorting algorithms: Bubble/Selection/Insertion Sort, etc. These algorithms are (awfully) slow and usually avoided in programming contests, though understanding them might help you solve certain problems.

2. $O(n \log n)$ comparison-based sorting algorithms: Merge/Heap/Quick Sort, etc. These algorithms are the default choice in programming contests as an $O(n \log n)$ complexity is optimal for comparison-based sorting. Therefore, these sorting algorithms run in the 'best possible' time in most cases (see below for special purpose sorting algorithms). In addition, these algorithms are well-known and hence we do not need to 'reinvent the wheel'[5]—we can simply use `sort`, `partial_sort`, or `stable_sort` in C++ STL `algorithm` (or `Collections.sort` in Java) for standard sorting tasks. We only need to specify the required comparison function and these library routines will handle the rest.

3. Special purpose sorting algorithms: $O(n)$ Counting/Radix/Bucket Sort, etc. Although rarely used, these special purpose algorithms are good to know as they can reduce the required sorting time if the data has certain special characteristics. For example, Counting Sort can be applied to integer data that lies in a small range (see Section 9.32).

---

[5]However, sometimes we do need to 'reinvent the wheel' for certain sorting-related problems, e.g. the Inversion Index problem in Section 9.14.

There are generally three common methods to search for an item in an array:

1. $O(n)$ Linear Search: Consider every element from index 0 to index $n-1$ (avoid this whenever possible).

2. $O(\log n)$ Binary Search: Use `lower_bound`, `upper_bound`, or `binary_search` in C++ STL `algorithm` (or Java `Collections.binarySearch`). If the input array is unsorted, it is necessary to sort the array at least once (using one of the $O(n \log n)$ sorting algorithm above) before executing one (or *many*) Binary Search(es).

3. $O(1)$ with Hashing: This is a useful technique to use when fast access to known values are required. If a suitable hash function is selected, the probability of a collision to be made is negligibly small. Still, this technique is rarely used and we can live without it[6] for most (contest) problems.

Visualization: `www.comp.nus.edu.sg/~stevenha/visualization/sorting.html`
Source code: `ch2_02_algorithm_collections.cpp/java`

- Array of Booleans: C++ STL `bitset` (Java `BitSet`)

If our array needs only to contain Boolean values (1/true and 0/false), we can use an alternative data structure other than an array—a C++ STL `bitset`. The `bitset` supports useful operations like `reset()`, `set()`, the `[]` operator and `test()`.

Source code: `ch5_06_primes.cpp/java`, also see Section 5.5.1

- Bitmasks a.k.a. lightweight, small sets of Booleans (native support in C/C++/Java)
An integer is stored in a computer's memory as a sequence/string of bits. Thus, we can use integers to represent a *lightweight* small set of Boolean values. All set operations then involve only the bitwise manipulation of the corresponding integer, which makes it a *much more efficient* choice when compared with the C++ STL `vector<bool>`, `bitset`, or `set<int>` options. Such speed is important in competitive programming. *Some* important operations that are used in this book are shown below.

```
Message: Check if j-th bit (from right) of S is on
                             {F D B }(set)
S=42 (dec)                 = 101010 (bin)
j=3, 1<<j=8 (dec)          = 001000 (bin)
                             ------ AND
T=8 (dec) =                  001000 (bin)
                           {   D   }(set)
```


S = [42]   (set all n = [-]   bits) | [LSOne] [set] [check] [clear] [toggle] | j = [3]   [shift left] [shift right]

Figure 2.1: Bitmask Visualization

1. Representation: A 32 (or 64)-bit *signed* integer for up to 32 (or 64) items[7]. Without a loss of generality, all examples below use a 32-bit signed integer called S.

---

[6]However, questions about hashing frequently appear in interviews for IT jobs.
[7]To avoid issues with the two's complement representation, use a 32-bit/64-bit *signed* integer to represent bitmasks of up to 30/62 items only, respectively.

```
Example:               5| 4| 3| 2| 1| 0 <- 0-based indexing from right
                      32|16| 8| 4| 2| 1 <- power of 2
S = 34 (base 10) = 1| 0| 0| 0| 1| 0 (base 2)
                       F| E| D| C| B| A <- alternative alphabet label
```

In the example above, the integer $S = 34$ or `100010` in binary also represents a small set $\{1, 5\}$ with a 0-based indexing scheme in increasing digit significance (or $\{B, F\}$ using the alternative alphabet label) because the second and the sixth bits (counting from the right) of S are on.

2. To multiply/divide an integer by 2, we only need to shift the bits in the integer left/right, respectively. This operation (especially the shift left operation) is important for the next few examples below. Notice that the truncation in the shift right operation automatically rounds the division-by-2 down, e.g. $17/2 = 8$.

```
S                     = 34 (base 10) =  100010 (base 2)
S = S << 1 = S * 2 = 68 (base 10) = 1000100 (base 2)
S = S >> 2 = S / 4 = 17 (base 10) =   10001 (base 2)
S = S >> 1 = S / 2 =  8 (base 10) =    1000 (base 2) <- LSB is gone
                                       (LSB = Least Significant Bit)
```

3. To set/turn on the j-th item (0-based indexing) of the set,
   use the bitwise OR operation `S |= (1 << j)`.

```
S = 34 (base 10) = 100010 (base 2)
j = 3, 1 << j    = 001000 <- bit '1' is shifted to the left 3 times
                   -------- OR (true if either of the bits is true)
S = 42 (base 10) = 101010 (base 2) // update S to this new value 42
```

4. To check if the j-th item of the set is on,
   use the bitwise AND operation `T = S & (1 << j)`.
   If `T = 0`, then the j-th item of the set is off.
   If `T != 0` (to be precise, `T = (1 << j)`), then the j-th item of the set is on.
   See Figure 2.1 for one such example.

```
S = 42 (base 10) = 101010 (base 2)
j = 3, 1 << j    = 001000 <- bit '1' is shifted to the left 3 times
                   -------- AND (only true if both bits are true)
T = 8 (base 10)  = 001000 (base 2) -> not zero, the 3rd item is on

S = 42 (base 10) = 101010 (base 2)
j = 2, 1 << j    = 000100 <- bit '1' is shifted to the left 2 times
                   -------- AND
T = 0 (base 10)  = 000000 (base 2) -> zero, the 2rd item is off
```

5. To clear/turn off the j-th item of the set,
   use[8] the bitwise AND operation `S &= ~(1 << j)`.

```
S = 42 (base 10) = 101010 (base 2)
j = 1, ~(1 << j) = 111101 <- '~' is the bitwise NOT operation
                   -------- AND
S = 40 (base 10) = 101000 (base 2) // update S to this new value 40
```

---

[8]Use brackets a lot when doing bit manipulation to avoid accidental bugs due to operator precedence.

6. To toggle (flip the status of) the j-th item of the set,
   use the bitwise XOR operation `S ^= (1 << j)`.

```
S = 40 (base 10) = 101000 (base 2)
j = 2, (1 << j)  = 000100 <- bit '1' is shifted to the left 2 times
                   -------- XOR <- true if both bits are different
S = 44 (base 10) = 101100 (base 2) // update S to this new value 44

S = 40 (base 10) = 101000 (base 2)
j = 3, (1 << j)  = 001000 <- bit '1' is shifted to the left 3 times
                   -------- XOR <- true if both bits are different
S = 32 (base 10) = 100000 (base 2) // update S to this new value 32
```

7. To get the value of the least significant bit that is on (first from the right),
   use `T = (S & (-S))`.

```
 S   =  40 (base 10) =  000...000101000 (32 bits, base 2)
-S   = -40 (base 10) =  111...111011000 (two's complement)
                        ----------------- AND
 T   =   8 (base 10) =  000...000001000 (3rd bit from right is on)
```

8. To turn on *all* bits in a set of size n, use `S = (1 << n) - 1`
   (be careful with overflows).

```
Example for n = 3
S + 1 = 8 (base 10) = 1000 <- bit '1' is shifted to left 3 times
                         1
                      ------ -
S     = 7 (base 10) =  111 (base 2)

Example for n = 5
S + 1 = 32 (base 10) = 100000 <- bit '1' is shifted to left 5 times
                          1
                       -------- -
S     = 31 (base 10) =  11111 (base 2)
```

Visualization: `www.comp.nus.edu.sg/~stevenha/visualization/bitmask.html`

Source code: `ch2_03_bit_manipulation.cpp/java`

Many bit manipulation operations are written as preprocessor macros in our C/C++ example source code (but written plainly in our Java example code since Java does not support macros).

- Linked List: C++ STL `list` (Java `LinkedList`)
  Although this data structure almost always appears in data structure and algorithm textbooks, the Linked List is usually avoided in typical (contest) problems. This is due to the inefficiency in accessing elements (a linear scan has to be performed from the head or the tail of a list) and the usage of pointers makes it prone to runtime errors if not implemented properly. In this book, almost all forms of Linked List have been replaced by the more flexible C++ STL `vector` (Java `Vector`).

The only exception is probably UVa 11988 - Broken Keyboard (a.k.a. Beiju Text)—where you are required to dynamically maintain a (linked) list of characters and efficiently insert a new character *anywhere* in the list, i.e. at front (head), current, or back (tail) of the (linked) list. Out of 1903 UVa problems that the authors have solved, this is likely to be the only pure linked list problem we have encountered thus far.

- Stack: C++ STL `stack` (Java `Stack`)
  This data structure is often used as part of algorithms that solve certain problems (e.g. bracket matching in Section 9.4, Postfix calculator and Infix to Postfix conversion in Section 9.27, finding Strongly Connected Components in Section 4.2.9 and Graham's scan in Section 7.3.7). A stack only allows for $O(1)$ insertion (push) and $O(1)$ deletion (pop) from the top. This behavior is usually referred to as Last In First Out (LIFO) and is reminiscent of literal stacks in the real world. Typical C++ STL `stack` operations include `push()`/`pop()` (insert/remove from top of stack), `top()` (obtain content from the top of stack), and `empty()`.

- Queue: C++ STL `queue` (Java `Queue`[9])
  This data structure is used in algorithms like Breadth First Search (BFS) in Section 4.2.2. A queue only allows for $O(1)$ insertion (enqueue) from the back (tail) and $O(1)$ deletion (dequeue) from the front (head). This behavior is similarly referred to as First In First Out (FIFO), just like actual queues in the real world. Typical C++ STL `queue` operations include `push()`/`pop()` (insert from back/remove from front of queue), `front()`/`back()` (obtain content from the front/back of queue), and `empty()`.

- Double-ended Queue (Deque): C++ STL `deque` (Java `Deque`[10])
  This data structure is very similar to the resizeable array (vector) and queue above, except that deques support fast $O(1)$ insertions and deletions at *both* the beginning and the end of the deque. This feature is important in certain algorithm, e.g. the Sliding Window algorithm in Section 9.31. Typical C++ STL `deque` operations include `push_back()`, `pop_front()` (just like the normal queue), `push_front()` and `pop_back()` (specific for deque).

Visualization: `www.comp.nus.edu.sg/~stevenha/visualization/list.html`

Source code: `ch2_04_stack_queue.cpp/java`

Exercise 2.2.1*: Suppose you are given an *unsorted* array S of n integers. Solve each of the following tasks below with the best possible algorithms that you can think of and analyze their time complexities. Let's assume the following constraints: $1 \leq n \leq 100K$ so that $O(n^2)$ solutions are theoretically infeasible in a contest environment.

1. Determine if S contains one or more pairs of duplicate integers.

2*. Given an integer v, find two integers $a, b \in S$ such that $a + b = v$.

3*. Follow-up to Question 2: what if the given array S is *already sorted*?

4*. Print the integers in S that fall between a range $[a \ldots b]$ (inclusive) in sorted order.

5*. Determine the length of the longest increasing *contiguous* sub-array in S.

6. Determine the median (50th percentile) of S. Assume that n is odd.

---

[9]The Java `Queue` is only an *interface* that is usually instantiated with Java `LinkedList`.

[10]The Java `Deque` is also an *interface*. `Deque` is usually instantiated with Java `LinkedList`.

Exercise 2.2.2: There are several other 'cool' tricks possible with bit manipulation techniques but these are rarely used. Please implement these tasks with bit manipulation:

1. Obtain the remainder (modulo) of S when it is divided by N (N is a power of 2)
   e.g. $S = (7)_{10} \% (4)_{10} = (111)_2 \% (100)_2 = (11)_2 = (3)_{10}$.

2. Determine if S is a power of 2.
   e.g. $S = (7)_{10} = (111)_2$ is not a power of 2, but $(8)_{10} = (100)_2$ is a power of 2.

3. Turn off the last bit in S, e.g. $S = (40)_{10} = (10\underline{1}000)_2 \rightarrow S = (32)_{10} = (10\underline{0}000)_2$.

4. Turn on the last zero in S, e.g. $S = (41)_{10} = (1010\underline{0}1)_2 \rightarrow S = (43)_{10} = (1010\underline{1}1)_2$.

5. Turn off the last consecutive run of ones in S
   e.g. $S = (39)_{10} = (100\underline{111})_2 \rightarrow S = (32)_{10} = (100\underline{000})_2$.

6. Turn on the last consecutive run of zeroes in S
   e.g. $S = (36)_{10} = (100\underline{100})_2 \rightarrow S = (39)_{10} = (100\underline{111})_2$.

7*. Solve UVa 11173 - Grey Codes with a *one-liner* bit manipulation expression for each test case, i.e. find the k-th Gray code.

8*. Let's reverse the UVa 11173 problem above. Given a gray code, find its position k using bit manipulation.

Exercise 2.2.3*: We can also use a *resizeable* array (C++ STL `vector` or Java `Vector`) to implement an efficient stack. Figure out how to achieve this. Follow up question: Can we use a *static* array, linked list, or deque instead? Why or why not?

Exercise 2.2.4*: We can use a linked list (C++ STL `list` or Java `LinkedList`) to implement an efficient queue (or deque). Figure out how to achieve this. Follow up question: Can we use a resizeable array instead? Why or why not?

Programming exercises involving linear data structures (and algorithms) with libraries:

- 1D Array Manipulation, e.g. array, C++ STL `vector` (or Java `Vector`/`ArrayList`)
    1. *UVa 00230 - Borrowers* (a bit of string parsing, see Section 6.2; maintain list of sorted books; sort key: author names first and if ties, by title; the input size is small although not stated; we do not need to use balanced BST)
    2. UVa 00394 - Mapmaker (any n-dimensional array is stored in computer memory as a single dimensional array; follow the problem description)
    3. UVa 00414 - Machined Surfaces (get longest stretch of 'B's)
    4. UVa 00467 - Synching Signals (linear scan, 1D boolean flag)
    5. UVa 00482 - Permutation Arrays (you may need to use a string tokenizer—see Section 6.2—as the size of the array is not specified)
    6. UVa 00591 - Box of Bricks (sum all items; get the average; sum the total absolute differences of each item from the average divided by two)
    7. *UVa 00665 - False Coin* (use 1D boolean flags; all coins are initially potential false coins; if '=', all coins on the left and right are not false coins; if '<' or '>', all coins not on the left and right are not false coins; check if there is only one candidate false coin left at the end)
    8. UVa 00755 - 487-3279 (Direct Addressing Table; convert the letters except Q & Z to 2-9; keep '0'-'9' as 0-9; sort the integers; find duplicates if any)

9. **UVa 10038 - Jolly Jumpers *** (use 1D boolean flags to check $[1..n-1]$)
10. UVa 10050 - Hartals (1D boolean flag)
11. UVa 10260 - Soundex (Direct Addressing Table for soundex code mapping)
12. UVa 10978 - Let's Play Magic (1D string array manipulation)
13. *UVa 11093 - Just Finish it up* (linear scan, circular array, a bit challenging)
14. UVa 11192 - Group Reverse (character array)
15. UVa 11222 - Only I did it (use several 1D arrays to simplify this problem)
16. **UVa 11340 - Newspaper *** (DAT; see Hashing in Section 2.3)
17. UVa 11496 - Musical Loop (store data in 1D array, count the peaks)
18. UVa 11608 - No Problem (use three arrays: created; required; available)
19. UVa 11850 - Alaska (for each integer location from 0 to 1322; can Brenda reach (anywhere within 200 miles of) any charging stations?)
20. *UVa 12150 - Pole Position* (simple manipulation)
21. ***UVa 12356 - Army Buddies **** (similar to deletion in doubly linked lists, but we can still use a 1D array for the underlying data structure)

- 2D Array Manipulation

  1. UVa 00101 - The Blocks Problem ('stack' like simulation; but we need to access the content of each stack too, so it is better to use 2D array)
  2. UVa 00434 - Matty's Blocks (a kind of visibility problem in geometry, solvable with using 2D array manipulation)
  3. UVa 00466 - Mirror Mirror (core functions: rotate and reflect)
  4. UVa 00541 - Error Correction (count the number of '1's for each row/col; all of them must be even; if $\exists$ an error, check if it is on the same row and col)
  5. UVa 10016 - Flip-flop the Squarelotron (tedious)
  6. UVa 10703 - Free spots (use 2D boolean array of size 500×500)
  7. **UVa 10855 - Rotated squares *** (string array, 90° clockwise rotation)
  8. **UVa 10920 - Spiral Tap *** (simulate the process)
  9. UVa 11040 - Add bricks in the wall (non trivial 2D array manipulation)
  10. UVa 11349 - Symmetric Matrix (use long long to avoid issues)
  11. UVa 11360 - Have Fun with Matrices (do as asked)
  12. **UVa 11581 - Grid Successors *** (simulate the process)
  13. UVa 11835 - Formula 1 (do as asked)
  14. *UVa 12187 - Brothers* (simulate the process)
  15. *UVa 12291 - Polyomino Composer* (do as asked, a bit tedious)
  16. *UVa 12398 - NumPuzz I* (simulate backwards, do not forget to mod 10)

- C++ STL `algorithm` (Java `Collections`)

  1. UVa 00123 - Searching Quickly (modified comparison function, use `sort`)
  2. **UVa 00146 - ID Codes *** (use `next_permutation`)
  3. UVa 00400 - Unix ls (this command very frequently used in UNIX)
  4. UVa 00450 - Little Black Book (tedious sorting problem)
  5. *UVa 00790 - Head Judge Headache* (similar to UVa 10258)
  6. UVa 00855 - Lunch in Grid City (sort, median)
  7. UVa 01209 - Wordfish (LA 3173, Manila06) (STL `next` and `prev_permutation`)
  8. *UVa 10057 - A mid-summer night ...* (involves the median, use STL `sort`, `upper_bound`, `lower_bound` and some checks)

9. **UVa 10107 - What is the Median? *** (find median of a *growing*/dynamic list of integers; still solvable with multiple calls of `nth_element` in `algorithm`)
10. UVa 10194 - Football a.k.a. Soccer (multi-fields sorting, use `sort`)
11. **UVa 10258 - Contest Scoreboard *** (multi-fields sorting, use `sort`)
12. *UVa 10698 - Football Sort* (multi-fields sorting, use `sort`)
13. UVa 10880 - Colin and Ryan (use `sort`)
14. UVa 10905 - Children's Game (modified comparison function, use `sort`)
15. UVa 11039 - Building Designing (use `sort` then count different signs)
16. UVa 11321 - Sort Sort and Sort (be careful with negative mod!)
17. UVa 11588 - Image Coding (`sort` simplifies the problem)
18. UVa 11777 - Automate the Grades (`sort` simplifies the problem)
19. UVa 11824 - A Minimum Land Price (`sort` simplifies the problem)
20. *UVa 12541 - Birthdates* (LA6148, HatYai12, `sort`, pick youngest and oldest)

- Bit Manipulation (both C++ STL `bitset` (Java `BitSet`) and bitmask)

  1. UVa 00594 - One Little, Two Little ... (manipulate bit string with `bitset`)
  2. UVa 00700 - Date Bugs (can be solved with `bitset`)
  3. UVa 01241 - Jollybee Tournament (LA 4147, Jakarta08, easy)
  4. **UVa 10264 - The Most Potent Corner *** (heavy bitmask manipulation)
  5. *UVa 11173 - Grey Codes* (D & C pattern or one liner bit manipulation)
  6. UVa 11760 - Brother Arif, ... (separate row+col checks; use two bitsets)
  7. **UVa 11926 - Multitasking *** (use 1M `bitset` to check if a slot is free)
  8. **UVa 11933 - Splitting Numbers *** (an exercise for bit manipulation)
  9. IOI 2011 - Pigeons (this problem becomes simpler with bit manipulation but the final solution requires much more than that.)

- C++ STL `list` (Java `LinkedList`)

  1. **UVa 11988 - Broken Keyboard ... *** (rare linked list problem)

- C++ STL `stack` (Java `Stack`)

  1. UVa 00127 - "Accordian" Patience (shuffling `stack`)
  2. **UVa 00514 - Rails *** (use `stack` to simulate the process)
  3. **UVa 00732 - Anagram by Stack *** (use `stack` to simulate the process)
  4. **UVa 01062 - Containers *** (LA 3752, WorldFinals Tokyo07, simulation with `stack`; maximum answer is 26 stacks; $O(n)$ solution exists)
  5. UVa 10858 - Unique Factorization (use `stack` to help solving this problem) Also see: implicit `stack`s in recursive function calls and Postfix conversion/evaluation in Section 9.27.

- C++ STL `queue` and `deque` (Java `Queue` and `Deque`)

  1. UVa 00540 - Team Queue (modified 'queue')
  2. **UVa 10172 - The Lonesome Cargo ... *** (use both `queue` and `stack`)
  3. **UVa 10901 - Ferry Loading III *** (simulation with `queue`)
  4. UVa 10935 - Throwing cards away I (simulation with `queue`)
  5. **UVa 11034 - Ferry Loading IV *** (simulation with `queue`)
  6. *UVa 12100 - Printer Queue* (simulation with `queue`)
  7. *UVa 12207 - This is Your Queue* (use both `queue` and `deque`) Also see: `queue`s in BFS (see Section 4.2.2)

2.3 Non-Linear DS with Built-in Libraries

For some problems, linear storage is not the best way to organize data. With the efficient implementations of non-linear data structures shown below, you can operate on the data in a quicker fashion, thereby speeding up the algorithms that rely on them.

For example, if you need a *dynamic*[11] collection of pairs (e.g. key → value pairs), using C++ STL `map` below can provide you $O(\log n)$ performance for insertion/search/deletion operations with just a few lines of code (that you still have to write yourself), whereas storing the same information inside a static array of `structs` may require $O(n)$ insertion/search/deletion, and you will need to write the longer traversal code yourself.

- Balanced Binary Search Tree (BST): C++ STL `map`/`set` (Java `TreeMap`/`TreeSet`)
  The BST is one way to organize data in a tree structure. In each subtree rooted at x, the following BST property holds: Items on the left subtree of x are smaller than x and items on the right subtree of x are greater than (or equal to) x. This is essentially an application of the Divide and Conquer strategy (also see Section 3.3). Organizing the data like this (see Figure 2.2) allows for $O(\log n)$ `search(key)`, `insert(key)`, `findMin()`/`findMax()`, `successor(key)`/`predecessor(key)`, and `delete(key)` since in the worst case, only $O(\log n)$ operations are required in a root-to-leaf scan (see [7, 5, 54, 12] for details). However, this only holds if the BST is balanced.
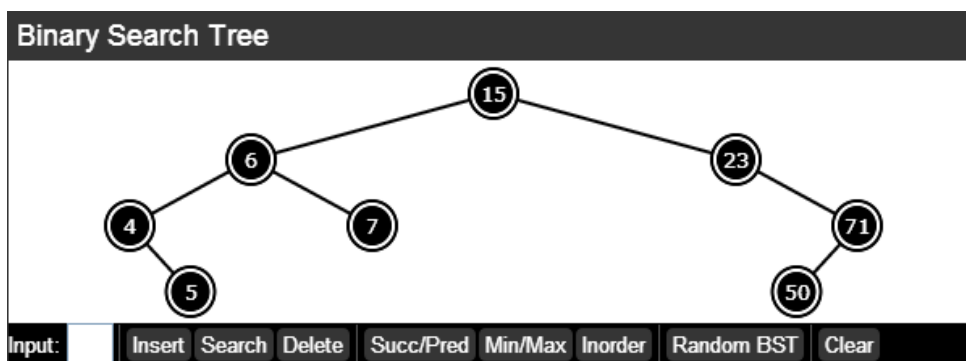
Figure 2.2: Examples of BST

Implementing *bug-free* balanced BSTs such as the Adelson-Velskii Landis (AVL)[12] or Red-Black (RB)[13] Trees is a tedious task and is difficult to achieve in a time-constrained contest environment (unless you have prepared a code library beforehand, see Section 9.29). Fortunately, C++ STL has `map` and `set` (and Java has `TreeMap` and `TreeSet`) which are *usually* implementations of the RB Tree which guarantees that major BST operations like insertions/searches/deletions are done in $O(\log n)$ time. By mastering these two C++ STL template classes (or Java APIs), you can save a lot of precious coding time during contests! The difference between these two data structures is simple: the C++ STL `map` (and Java `TreeMap`) stores (key → data) pairs whereas the C++

---

[11]The contents of a dynamic data structure is frequently modified via insert/delete/update operations.

[12]The AVL tree was the first self-balancing BST to be invented. AVL trees are essentially traditional BSTs with an additional property: The heights of the two subtrees of any vertex in an AVL tree can differ *by at most one*. Rebalancing operations (rotations) are performed (when necessary) during insertions and deletions to maintain this invariant property, hence keeping the tree roughly balanced.

[13]The Red-Black tree is another self-balancing BST, in which every vertex has a color: red or black. In RB trees, the root vertex, all leaf vertices, and both children of every red vertex are black. Every simple path from a vertex to any of its descendant leaves contains *the same number of black vertices*. Throughout insertions and deletions, an RB tree will maintain all these invariants to keep the tree balanced.

STL `set` (and Java `TreeSet`) only stores the key. For most (contest) problems, we use a `map` (to really map information) instead of a `set` (a `set` is only useful for efficiently determining the existence of a certain key). However, there is a small drawback. If we use the library implementations, it becomes difficult or impossible to augment (add extra information to) the BST. Please attempt **Exercise 2.3.5*** and read Section 9.29 for more details.

Visualization: `www.comp.nus.edu.sg/~stevenha/visualization/bst.html`

Source code: `ch2_05_map_set.cpp/java`

- Heap: C++ STL `priority_queue` (Java `PriorityQueue`)
  The heap is another way to organize data in a tree. The (Binary) Heap is also a binary tree like the BST, except that it must be a *complete*[14] tree. Complete binary trees can be stored efficiently in a compact 1-indexed array of size $n + 1$, which is often preferred to an explicit tree representation. For example, the array A = {N/A, 90, 19, 36, 17, 3, 25, 1, 2, 7} is the compact array representation of Figure 2.3 with index 0 ignored. One can navigate from a certain index (vertex) i to its parent, left child, and right child by using simple index manipulation: $\lfloor \frac{i}{2} \rfloor$, $2 \times i$, and $2 \times i + 1$, respectively. These index manipulations can be made faster using bit manipulation techniques (see Section 2.2): `i >> 1`, `i << 1`, and `(i << 1) + 1`, respectively.

Instead of enforcing the BST property, the (Max) Heap enforces the Heap property: in each subtree rooted at x, items on the left **and** right subtrees of x are smaller than (or equal to) x (see Figure 2.3). This is also an application of the Divide and Conquer concept (see Section 3.3). The property guarantees that the top (or root) of the heap is always the maximum element. There is no notion of a 'search' in the Heap (unlike BSTs). The Heap instead allows for the fast extraction (deletion) of the maximum element: `ExtractMax()` and insertion of new items: `Insert(v)`—both of which can be easily achieved by in a $O(\log n)$ root-to-leaf or leaf-to-root traversal, performing swapping operations to maintain the (Max) Heap property whenever necessary (see [7, 5, 54, 12] for details).
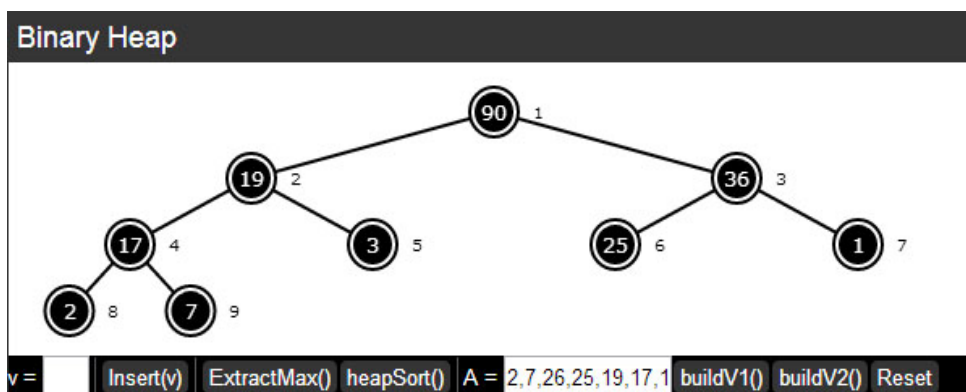
Figure 2.3: (Max) Heap Visualization

The (Max) Heap is a useful data structure for modeling a Priority Queue, where the item with the highest priority (the maximum element) can be dequeued (`ExtractMax()`)

---

[14]A complete binary tree is a binary tree in which every level, except possibly the last, is completely filled. All vertices in the last level must also be filled from left-to-right.

and a new item v can be enqueued (`Insert(v)`), both in $O(\log n)$ time. The implementation[15] of `priority_queue` is available in the C++ STL `queue` library (or Java `PriorityQueue`). Priority Queues are an important component in algorithms like Prim's (and Kruskal's) algorithms for the Minimum Spanning Tree (MST) problem (see Section 4.3), Dijkstra's algorithm for the Single-Source Shortest Paths (SSSP) problem (see Section 4.4.3), and the A* Search algorithm (see Section 8.2.5).

This data structure is also used to perform `partial_sort` in the C++ STL `algorithm` library. One possible implementation is by processing the elements one by one and creating a Max[16] Heap of k elements, removing the largest element whenever its size exceeds k (k is the number of elements requested by user). The smallest k elements can then be obtained in descending order by dequeuing the remaining elements in the Max Heap. As each dequeue operation is $O(\log k)$, `partial_sort` has $O(n \log k)$ time complexity[17]. When $k = n$, this algorithm is equivalent to a heap sort. Note that although the time complexity of a heap sort is also $O(n \log n)$, heap sorts are often slower than quick sorts because heap operations access data stored in distant indices and are thus not cache-friendly.

Visualization: `www.comp.nus.edu.sg/~stevenha/visualization/heap.html`

Source code: `ch2_06_priority_queue.cpp/java`

- Hash Table: C++11 STL `unordered_map`[18] (and Java `HashMap`/`HashSet`/`HashTable`)
  The Hash Table is another non-linear data structure, but we do not recommend using it in programming contests unless absolutely necessary. Designing a well-performing hash function is often tricky and only the new C++11 has STL support for it (Java has Hash-related classes).

  Moreover, C++ STL `maps` or `sets` (and Java `TreeMaps` or `TreeSets`) are usually fast enough as the typical input size of (programming contest) problems is usually not more than $1M$. Within these bounds, the $O(1)$ performance of Hash Tables and $O(\log 1M)$ performance for balanced BSTs do not differ by much. Thus, we do not discuss Hash Tables in detail in this section.

  However, a simple form of Hash Tables can be used in programming contests. 'Direct Addressing Tables' (DATs) can be considered to be Hash Tables where the keys themselves are the indices, or where the 'hash function' is the identity function. For example, we may need to assign all possible ASCII characters [0-255] to integer values, e.g. 'a' → '3', 'W' → '10', ..., 'I' → '13'. For this purpose, we do not need the C++ STL `map` or any form of hashing as the key itself (the value of the ASCII character) is unique and sufficient to determine the appropriate index in an array of size 256. Some programming exercises involving DATs are listed in the previous Section 2.2.

---

[15]The default C++ STL `priority_queue` is a Max Heap (dequeuing yields items in descending key order) whereas the default Java `PriorityQueue` is a Min Heap (yields items in ascending key order). Tips: A Max Heap containing numbers can easily be converted into a Min Heap (and vice versa) by inserting the negated keys. This is because negating a set of numbers will reverse their order of appearance when sorted. This trick is used several times in this book. However, if the priority queue is used to store *32-bit signed integers*, an overflow will occur if -2^{31} is negated as $2^{31} - 1$ is the maximum value of a 32-bit signed integer.

[16]The default `partial_sort` produces the smallest k elements in ascending order.

[17]You may have noticed that the time complexity $O(n \log k)$ where k is the output size and n is the input size. This means that the algorithm is 'output-sensitive' since its running time depends not only on the input size but also on the amount of items that it has to output.

[18]Note that C++11 is a new C++ standard, older compilers may not support it yet.

Exercise 2.3.1: Someone suggested that it is possible to store the key → value pairs in a *sorted array* of `structs` so that we can use the $O(\log n)$ binary search for the example problem above. Is this approach feasible? If no, what is the issue?

Exercise 2.3.2: We will not discuss the basics of BST operations in this book. Instead, we will use a series of sub-tasks to verify your understanding of BST-related concepts. We will use Figure 2.2 as an *initial reference* in all sub-tasks except sub-task 2.

1. Display the steps taken by `search(71)`, `search(7)`, and then `search(22)`.

2. Starting with an *empty* BST, display the steps taken by `insert(15)`, `insert(23)`, `insert(6)`, `insert(71)`, `insert(50)`, `insert(4)`, `insert(7)`, and `insert(5)`.

3. Display the steps taken by `findMin()` (and `findMax()`).

4. Indicate the *inorder traversal* of this BST. Is the output sorted?

5. Display the steps taken by `successor(23)`, `successor(7)`, and `successor(71)`.

6. Display the steps taken by `delete(5)` (a leaf), `delete(71)` (an internal node with one child), and then `delete(15)` (an internal node with two children).

Exercise 2.3.3*: Suppose you are given a reference to the root R of a binary tree T containing n vertices. You can access a node's left, right and parent vertices as well as its key through its reference. Solve each of the following tasks below with the best possible algorithms that you can think of and analyze their time complexities. Let's assume the following constraints: $1 \le n \le 100K$ so that $O(n^2)$ solutions are theoretically infeasible in a contest environment.

1. Check if T is a BST.

2*. Output the elements in T that are within a given range $[a..b]$ in ascending order.

3*. Output the contents of the *leaves* of T in *descending order*.

Exercise 2.3.4*: The inorder traversal (also see Section 4.7.2) of a standard (not necessarily balanced) BST is known to produce the BST's element in sorted order and runs in $O(n)$. Does the code below also produce the BST elements in sorted order? Can it be made to run in a total time of $O(n)$ instead of $O(\log n + (n-1) \times \log n) = O(n \log n)$? If possible, how?

```
x = findMin(); output x
for (i = 1; i < n; i++)                    // is this loop O(n log n)?
  x = successor(x); output x
```

Exercise 2.3.5*: Some (hard) problems require us to write *our own* balanced Binary Search Tree (BST) implementations due to the need to augment the BST with additional data (see Chapter 14 of [7]). Challenge: Solve UVa 11849 - CD which is a pure balanced BST problem with *your own* balanced BST implementation to test its performance and correctness.

Exercise 2.3.6: We will not discuss the basics of Heap operations in this book. Instead, we will use a series of questions to verify your understanding of Heap concepts.

1. With Figure 2.3 as the initial heap, display the steps taken by `Insert(26)`.

2. After answering question 1 above, display the steps taken by `ExtractMax()`.

Exercise 2.3.7: Is the structure represented by a 1-based compact array (ignoring index 0) sorted in descending order a Max Heap?

Exercise 2.3.8*: Prove or disprove this statement: "The second largest element in a Max Heap with $n \geq 3$ distinct elements is always one of the direct children of the root". Follow up question: What about the third largest element? Where is/are the potential location(s) of the third largest element in a Max Heap?

Exercise 2.3.9*: Given a 1-based compact array A containing n integers ($1 \leq n \leq 100K$) that are guaranteed to satisfy the Max Heap property, output the elements in A that are greater than an integer v. What is the best algorithm?

Exercise 2.3.10*: Given an unsorted array S of n distinct integers ($2k \leq n \leq 100000$), find the largest and smallest k ($1 \leq k \leq 32$) integers in S in $O(n \log k)$. Note: For this written exercise, assume that an $O(n \log n)$ algorithm is *not* acceptable.

Exercise 2.3.11*: One heap operation *not* directly supported by the C++ STL `priority_queue` (and Java `PriorityQueue`) is the `UpdateKey(index, newKey)` operation, which allows the (Max) Heap element at a certain index to be updated (increased or decreased). Write *your own* binary (Max) Heap implementation with this operation.

Exercise 2.3.12*: Another heap operation that may be useful is the `DeleteKey(index)` operation to delete (Max) Heap elements at a certain index. Implement this!

Exercise 2.3.13*: Suppose that we only need the `DecreaseKey(index, newKey)` operation, i.e. an `UpdateKey` operation where the update *always* makes `newKey` smaller than its previous value. Can we use a simpler approach than in **Exercise 2.3.11**? Hint: Use lazy deletion, we will use this technique in our Dijkstra code in Section 4.4.3.

Exercise 2.3.14*: Is it possible to use a balanced BST (e.g. C++ STL `set` or Java `TreeSet`) to implement a Priority Queue with the same $O(\log n)$ enqueue and dequeue performance? If yes, how? Are there any potential drawbacks? If no, why?

Exercise 2.3.15*: Is there a better way to implement a Priority Queue if the keys are all integers within a small range, e.g. $[0 \ldots 100]$? We are expecting an $O(1)$ enqueue and dequeue performance. If yes, how? If no, why?

Exercise 2.3.16: Which non-linear data structure should you use if you have to support the following three dynamic operations: 1) many insertions, 2) many deletions, and 3) many requests for the data in sorted order?

Exercise 2.3.17: There are M **strings**. N of them are unique ($N \leq M$). Which non-linear data structure discussed in this section should you use if you have to index (label) these **M** strings with integers from `[0..N-1]`? The indexing criteria is as follows: The first string must be given an index of 0; The next different string must be given index 1, and so on. However, if a string is re-encountered, it must be given the same index as its earlier copy! One application of this task is in constructing the connection graph from a list of city names (which are not integer indices!) and a list of highways between these cities (see Section 2.4.1). To do this, we first have to map these city names into integer indices (which are far more efficient to work with).

Programming exercises solvable with library of non-linear data structures:

- C++ STL `map` (and Java `TreeMap`)
    1. UVa 00417 - Word Index (generate all words, add to `map` for auto sorting)
    2. UVa 00484 - The Department of ... (maintain frequency with `map`)
    3. UVa 00860 - Entropy Text Analyzer (frequency counting)
    4. *UVa 00939 - Genes* (`map` child name to his/her gene and parents' names)
    5. *UVa 10132 - File Fragmentation* (N = number of fragments, B = total bits of all fragments divided by $N/2$; try all $2 \times N^2$ concatenations of two fragments that have length B; report the one with highest frequency; use `map`)
    6. *UVa 10138 - CDVII* (map plates to bills, entrace time and position)
    7. **UVa 10226 - Hardwood Species *** (use hashing for a better performance)
    8. UVa 10282 - Babelfish (a pure dictionary problem; use `map`)
    9. UVa 10295 - Hay Points (use `map` to deal with Hay Points dictionary)
    10. UVa 10686 - SQF Problem (use `map` to manage the data)
    11. UVa 11239 - Open Source (use `map` and `set` to check previous strings)
    12. **UVa 11286 - Conformity *** (use `map` to keep track of the frequencies)
    13. UVa 11308 - Bankrupt Baker (use `map` and `set` to help manage the data)
    14. *UVa 11348 - Exhibition* (use `map` and `set` to check uniqueness)
    15. ***UVa 11572 - Unique Snowflakes *** (use `map` to record the occurrence index of a certain snowflake size; use this to determine the answer in $O(n \log n)$)
    16. UVa 11629 - Ballot evaluation (use `map`)
    17. UVa 11860 - Document Analyzer (use `set` and `map`, linear scan)
    18. UVa 11917 - Do Your Own Homework (use `map`)
    19. *UVa 12504 - Updating a Dictionary* (use `map`; string to string; a bit tedious)
    20. *UVa 12592 - Slogan Learning of Princess* (use `map`; string to string)
    Also check frequency counting section in Section 6.3.

- C++ STL `set` (Java `TreeSet`)
    1. UVa 00501 - Black Box (use `multiset` with efficient iterator manipulation)
    2. **UVa 00978 - Lemmings Battle *** (simulation, use `multiset`)
    3. UVa 10815 - Andy's First Dictionary (use `set` and `string`)
    4. UVa 11062 - Andy's Second Dictionary (similar to UVa 10815, with twists)
    5. **UVa 11136 - Hoax or what *** (use `multiset`)
    6. **UVa 11849 - CD *** (use `set` to pass the time limit, better: use hashing!)
    7. *UVa 12049 - Just Prune The List* (`multiset` manipulation)

- C++ STL `priority_queue` (Java `PriorityQueue`)
    1. **UVa 01203 - Argus *** (LA 3135, Beijing04; use `priority_queue`)
    2. **UVa 10954 - Add All *** (use `priority_queue`, greedy)
    3. **UVa 11995 - I Can Guess ... *** (`stack`, `queue`, and `priority_queue`)
    Also see the usage of `priority_queue` for topological sorts (see Section 4.2.1), Kruskal's[19] (see Section 4.3.2), Prim's (see Section 4.3.3), Dijkstra's (see Section 4.4.3), and the A* Search algorithms (see Section 8.2.5)

---

[19]This is another way to implement the edge sorting in Kruskal's algorithm. Our (C++) implementation shown in Section 4.3.2 simply uses `vector` + `sort` instead of `priority_queue` (a heap sort).

2.4 Data Structures with Our Own Libraries

As of 24 May 2013, important data structures shown in this section do not have built-in support yet in C++ STL or Java API. Thus, to be competitive, contestants should prepare bug-free implementations of these data structures. In this section, we discuss the key ideas and example implementations (see the given source code too) of these data structures.

2.4.1 Graph

The graph is a pervasive structure which appears in many Computer Science problems. A graph $(G = (V, E))$ in its basic form is simply a set of vertices (V) and edges $(E$; storing connectivity information between vertices in V). Later in Chapter 3, 4, 8, and 9, we will explore many important graph problems and algorithms. To prepare ourselves, we will discuss three basic ways (there are a few other rare structures) to represent a graph G with V vertices and E edges in this subsection[20].
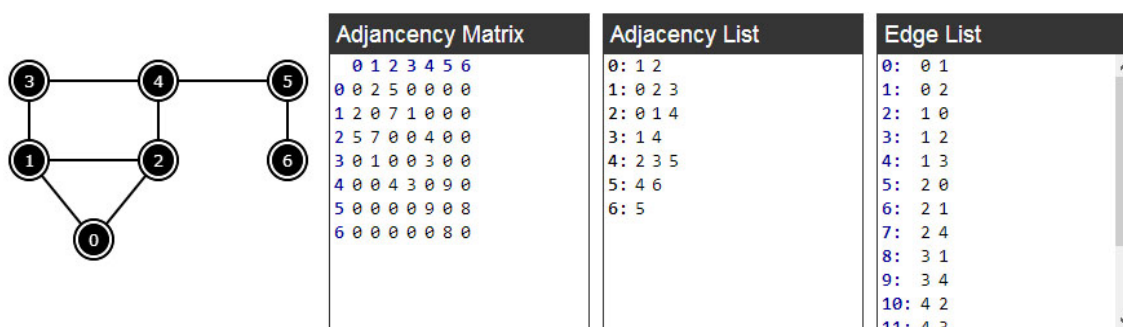
Figure 2.4: Graph Data Structure Visualization

A). The Adjacency Matrix, usually in the form of a 2D array (see Figure 2.4).

In (programming contest) problems involving graphs, the number of vertices V is usually known. Thus we can build a 'connectivity table' by creating a static 2D array: `int AdjMat[V][V]`. This has an $O(V^2)$ *space*[21] complexity. For an unweighted graph, set `AdjMat[i][j]` to a non-zero value (usually 1) if there is an edge between vertex *i-j* or zero otherwise. For a weighted graph, set `AdjMat[i][j] = weight(i,j)` if there is an edge between vertex *i-j* with `weight(i,j)` or zero otherwise. Adjacency Matrix cannot be used to store multigraph. For a simple graph without self-loops, the main diagonal of the matrix contains only zeroes, i.e. `AdjMat[i][i] = 0`, $\forall i \in$ `[0..V-1]`.

An Adjacency Matrix is a good choice if the connectivity between two vertices in a *small dense graph* is frequently required. However, it is not recommended for *large sparse graphs* as it would require too much space ($O(V^2)$) and there would be many blank (zero) cells in the 2D array. In a competitive setting, it is usually infeasible to use Adjacency Matrices when the given V is larger than ≈ 1000. Another drawback of Adjacency Matrix is that it also takes $O(V)$ time to enumerate the list of neighbors of a vertex v—an operation common to many graph algorithms—even if a vertex only has a handful of neighbors. A more compact and efficient graph representation is the Adjacency List discussed below.

---

[20]The most appropriate notation for the cardinality of a set S is $|S|$. However, in this book, we will often overload the meaning of V or E to also mean $|V|$ or $|E|$, depending on the context.

[21]We differentiate between the *space* and *time* complexities of data structures. The *space* complexity is an asymptotic measure of the memory requirements of a data structure whereas the *time* complexity is an asymptotic measure of the time taken to run a certain algorithm or an operation on the data structure.

B). The Adjacency List, usually in the form of a vector of vector of pairs (see Figure 2.4).
Using the C++ STL: `vector<vii> AdjList`, with `vii` defined as in:
`typedef pair<int, int> ii; typedef vector<ii> vii; // data type shortcuts`
Using the Java API: `Vector< Vector < IntegerPair > > AdjList`.
`IntegerPair` is a simple Java class that contains a pair of integers like `ii` above.

In Adjacency Lists, we have a `vector` of `vector` of pairs, storing the list of neighbors of each vertex u as 'edge information' pairs. Each pair contains two pieces of information, the index of the neighbouring vertex and the weight of the edge. If the graph is unweighted, simply store the weight as 0, 1, or drop the weight attribute[22] entirely. The space complexity of Adjacency List is $O(V + E)$ because if there are E bidirectional edges in a (simple) graph, this Adjacency List will only store $2E$ 'edge information' pairs. As E is usually much smaller than $V \times (V-1)/2 = O(V^2)$—the maximum number of edges in a complete (simple) graph, Adjacency Lists are often more space-efficient than Adjacency Matrices. Note that Adjacency List can be used to store multigraph.

With Adjacency Lists, we can also enumerate the list of neighbors of a vertex v efficiently. If v has k neighbors, the enumeration will require $O(k)$ time. Since this is one of the most common operations in most graph algorithms, it is advisable to use Adjacency Lists as your first choice of graph representation. Unless otherwise stated, most graph algorithms discussed in this book use the Adjacency List.

C). The Edge List, usually in the form of a vector of triples (see Figure 2.4).
Using the C++ STL: `vector< pair<int, ii> > EdgeList`.
Using the Java API: `Vector< IntegerTriple > EdgeList`.
`IntegerTriple` is a class that contains a triple of integers like `pair<int, ii>` above.

In the Edge List, we store a list of all E edges, usually in some sorted order. For directed graphs, we can store a bidirectional edge twice, one for each direction. The space complexity is clearly $O(E)$. This graph representation is very useful for Kruskal's algorithm for MST (Section 4.3.2), where the collection of undirected edges need to be sorted[23] by ascending weight. However, storing graph information in Edge List complicates many graph algorithms that require the enumeration of edges incident to a vertex.

Visualization: `www.comp.nus.edu.sg/~stevenha/visualization/graphds.html`

Source code: `ch2_07_graph_ds.cpp/java`

Implicit Graph

Some graphs do *not* have to be stored in a graph data structure or explicitly generated for the graph to be traversed or operated upon. Such graphs are called *implicit* graphs. You will encounter them in the subsequent chapters. Implicit graphs can come in two flavours:

1. The edges can be determined easily.

   Example 1: Navigating a 2D grid map (see Figure 2.5.A). The vertices are the cells in the 2D character grid where '.' represents land and '#' represents an obstacle. The edges can be determined easily: There is an edge between two neighboring cells in the

---

[22]For simplicity, we will always assume that the second attribute exists in all graph implementations in this book although it is not always used.

[23]`pair` objects in C++ can be easily sorted. The default sorting criteria is to sort on the first item and then the second item for tie-breaking. In Java, we can write our own `IntegerPair`/`IntegerTriple` class that implements `Comparable`.

grid if they share an N/S/E/W border and if both are '.' (see Figure 2.5.B).

Example 2: The graph of chess knight movements on an 8×8 chessboard. The vertices are the cells in the chessboard. Two squares in the chessboard have an edge between them if they differ by two squares horizontally and one square vertically (or two squares vertically and one square horizontally). The first three rows and four columns of a chessboard are shown in Figure 2.5.C (many other vertices and edges are not shown).

2. The edges can be determined with some rules.

Example: A graph contains N vertices ([1..N]). There is an edge between two vertices i and j if $(i+j)$ is a prime. See Figure 2.5.D that shows such a graph with $N = 5$ and several more examples in Section 8.2.3.
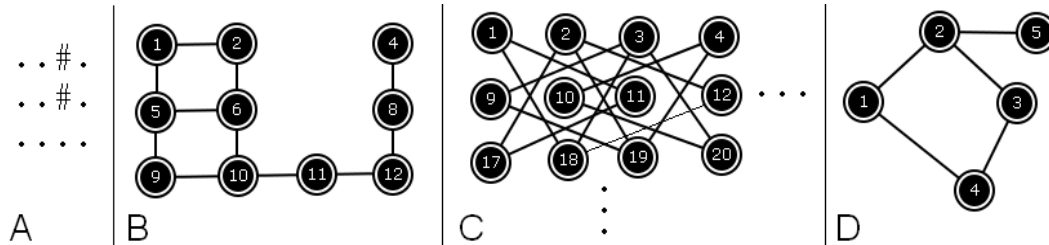
Figure 2.5: Implicit Graph Examples

Exercise 2.4.1.1*: Create the Adjacency Matrix, Adjacency List, and Edge List representations of the graphs shown in Figure 4.1 (Section 4.2.1) and in Figure 4.9 (Section 4.2.9). Hint: Use the graph data structure visualization tool shown above.

Exercise 2.4.1.2*: Given a (simple) graph in one representation (Adjacency Matrix/AM, Adjacency List/AL, or Edge List/EL), *convert* it into another graph representation in the most efficient way possible! There are 6 possible conversions here: AM to AL, AM to EL, AL to AM, AL to EL, EL to AM, and EL to AL.

Exercise 2.4.1.3: If the Adjacency Matrix of a (simple) graph has the property that it is equal to its transpose, what does this imply?

Exercise 2.4.1.4*: Given a (simple) graph represented by an Adjacency Matrix, perform the following tasks in the most efficient manner. Once you have figured out how to do this for Adjacency Matrices, perform the same task with Adjacency Lists and then Edge Lists.

1. Count the number of vertices V and directed edges E (assume that a bidirectional edge is equivalent to two directed edges) of the graph.

2*. Count the in-degree and the out-degree of a certain vertex v.

3*. Transpose the graph (reverse the direction of each edges).

4*. Check if the graph is a complete graph K_n. Note: A complete graph is a simple undirected graph in which *every pair* of distinct vertices is connected by a single edge.

5*. Check if the graph is a tree (a connected undirected graph with $E = V - 1$ edges).

6*. Check if the graph is a star graph S_k. Note: A star graph S_k is a complete bipartite $K_{1,k}$ graph. It is a tree with only one internal vertex and k leaves.

Exercise 2.4.1.5*: Research other possible methods of representing graphs other than the ones discussed above, especially for storing special graphs!

2.4.2 Union-Find Disjoint Sets

The Union-Find Disjoint Set (UFDS) is a data structure to model a collection of *disjoint sets* with the ability to efficiently[24]—in $\approx O(1)$—determine which set an item belongs to (or to test whether two items belong to the same set) and to unite two disjoint sets into one larger set. Such data structure can be used to solve the problem of finding connected components in an undirected graph (Section 4.2.3). Initialize each vertex to a separate disjoint set, then enumerate the graph's edges and join every two vertices/disjoint sets connected by an edge. We can then test if two vertices belong to the same component/set easily.

These seemingly simple operations are not *efficiently* supported by the C++ STL `set` (and Java `TreeSet`), which is not designed for this purpose. Having a `vector` of `set`s and looping through each one to find which set an item belongs to is expensive! C++ STL `set_union` (in `algorithm`) will not be efficient enough although it combines two sets in *linear time* as we still have to deal with shuffling the contents of the `vector` of `set`s! To support these set operations efficiently, we need a better data structure—the UFDS.

The main innovation of this data structure is in choosing a representative 'parent' item to represent a set. If we can ensure that each set is represented by only one unique item, then determining if items belong to the same set becomes far simpler: The representative 'parent' item can be used as a sort of identifier for the set. To achieve this, the Union-Find Disjoint Set creates a tree structure where the disjoint sets form a forest of trees. Each tree corresponds to a disjoint set. The root of the tree is determined to be the representative item for a set. Thus, the representative set identifier for an item can be obtained simply by following the chain of parents to the root of the tree, and since a tree can only have one root, this representative item can be used as a unique identifier for the set.

To do this efficiently, we store the index of the parent item and (the upper bound of) the height of the tree of each set (`vi p` and `vi rank` in our implementation). Remember that `vi` is our shortcut for a vector of integers. `p[i]` stores the immediate parent of item `i`. If item `i` is the representative item of a certain disjoint set, then `p[i] = i`, i.e. a self-loop. `rank[i]` yields (the upper bound of) the height of the tree rooted at item `i`.

In this section, we will use 5 disjoint sets {0, 1, 2, 3, 4} to illustrate the usage of this data structure. We initialize the data structure such that each item is a disjoint set by itself with rank 0 and the parent of each item is initially set to itself.

To unite two disjoint sets, we set the representative item (root) of one disjoint set to be the new parent of the representative item of the other disjoint set. This effectively merges the two trees in the Union-Find Disjoint Set representation. As such, `unionSet(i, j)` will cause both items 'i' and 'j' to have the same representative item—directly or indirectly. For efficiency, we can use the information contained in `vi rank` to set the representative item of the disjoint set with *higher rank* to be the new parent of the disjoint set with *lower rank*, thereby `minimizing` the rank of the resulting tree. If both ranks are the same, we arbitrarily choose one of them as the new parent and increase the resultant root's rank. This is the 'union by rank' heuristic. In Figure 2.6, top, `unionSet(0, 1)` sets `p[0]` to 1 and `rank[1]` to 1. In Figure 2.6, middle, `unionSet(2, 3)` sets `p[2]` to 3 and `rank[3]` to 1.

For now, let's assume that function `findSet(i)` simply calls `findSet(p[i])` recursively to find the representative item of a set, returning `findSet(p[i])` whenever `p[i] != i` and `i` otherwise. In Figure 2.6, bottom, when we call `unionSet(4, 3)`, we have `rank[findSet(4)]` = `rank[4]` = 0 which is smaller than `rank[findSet(3)]` = `rank[3]` = 1, so we set `p[4]` = 3 *without* changing the height of the resulting tree—this is the 'union by rank' heuristic

---

[24]M operations of this UFDS data structure with 'path compression' and 'union by rank' heuristics run in $O(M \times \alpha(n))$. However, since the inverse Ackermann function $\alpha(n)$ grows very slowly, i.e. its value is just less than 5 for practical input size $n \leq 1M$ in programming contest setting, we can treat $\alpha(n)$ as constant.
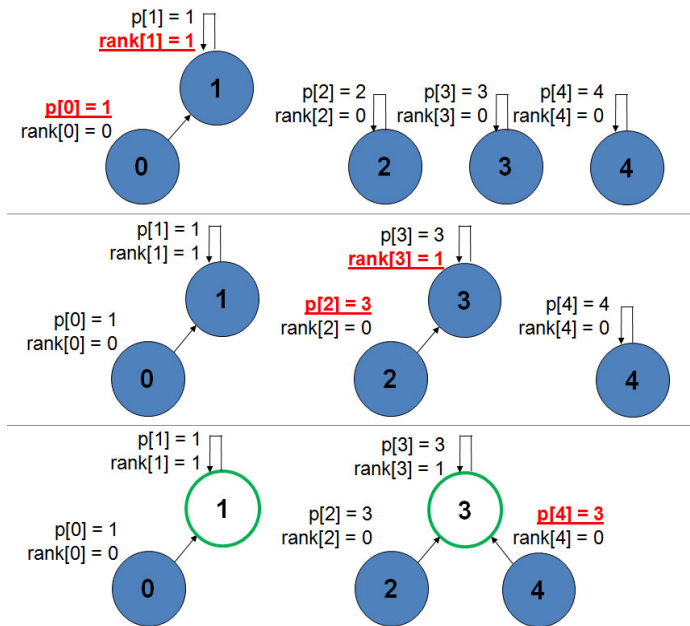
Figure 2.6: `unionSet(0, 1)` → `(2, 3)` → `(4, 3)` and `isSameSet(0, 4)`

at work. With the heuristic, the path taken from any node to the representative item by following the chain of 'parent' links is effectively minimized.

In Figure 2.6, bottom, `isSameSet(0, 4)` demonstrates another operation for this data structure. This function `isSameSet(i, j)` simply calls `findSet(i)` and `findSet(j)` and checks if both refer to the same representative item. If they do, then 'i' and 'j' both belong to the same set. Here, we see that `findSet(0) = findSet(p[0]) = findSet(1) = 1` is not the same as `findSet(4)= findSet(p[4]) = findSet(3) = 3`. Therefore item 0 and item 4 belongs to *different* disjoint sets.
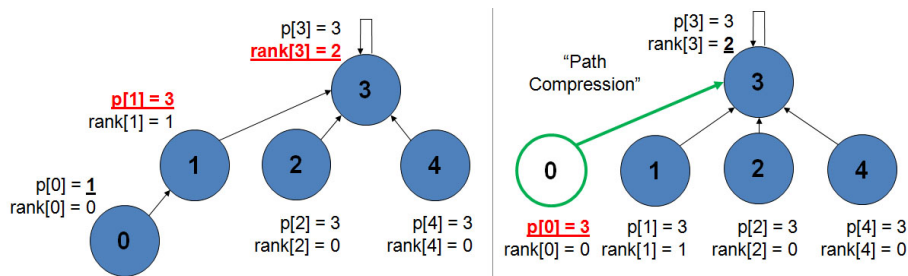
Figure 2.7: `unionSet(0, 3)` → `findSet(0)`

There is a technique that can vastly speed up the `findSet(i)` function: Path compression. Whenever we find the representative (root) item of a disjoint set by following the chain of 'parent' links from a given item, we can set the parent of *all items* traversed to point directly to the root. Any subsequent calls to `findSet(i)` on the affected items will then result in only one link being traversed. This changes the structure of the tree (to make `findSet(i)` more efficient) but yet preserves the actual constitution of the disjoint set. Since this will occur any time `findSet(i)` is called, the combined effect is to render the runtime of the `findSet(i)` operation to run in an extremely efficient amortized $O(M \times \alpha(n))$ time.

In Figure 2.7, we demonstrate this 'path compression'. First, we call `unionSet(0, 3)`. This time, we set `p[1] = 3` and update `rank[3] = 2`. Now notice that `p[0]` is unchanged, i.e. `p[0] = 1`. This is an *indirect* reference to the (true) representative item of the set, i.e. `p[0] = 1` → `p[1] = 3`. Function `findSet(i)` will actually require more than one step to

traverse the chain of 'parent' links to the root. However, once it finds the representative item, (e.g. 'x') for that set, it will *compress the path* by setting `p[i] = x`, i.e. `findSet(0)` sets `p[0] = 3`. Therefore, subsequent calls of `findSet(i)` will be just $O(1)$. This simple strategy is aptly named the 'path compression' heuristic. Note that `rank[3] = 2` now no longer reflects the *true height* of the tree. This is why `rank` only reflects the *upper bound* of the actual height of the tree. Our C++ implementation is shown below:

```cpp
class UnionFind {                                          // OOP style
private: vi p, rank;                        // remember: vi is vector<int>
public:
  UnionFind(int N) { rank.assign(N, 0);
    p.assign(N, 0); for (int i = 0; i < N; i++) p[i] = i; }
  int findSet(int i) { return (p[i] == i) ? i : (p[i] = findSet(p[i])); }
  bool isSameSet(int i, int j) { return findSet(i) == findSet(j); }
  void unionSet(int i, int j) {
    if (!isSameSet(i, j)) {                        // if from different set
      int x = findSet(i), y = findSet(j);
      if (rank[x] > rank[y]) p[y] = x;          // rank keeps the tree short
      else {                   p[x] = y;
                               if (rank[x] == rank[y]) rank[y]++; }
} } };
```

Visualization: `www.comp.nus.edu.sg/~stevenha/visualization/ufds.html`

Source code: `ch2_08_unionfind_ds.cpp/java`

Exercise 2.4.2.1: There are two more queries that are commonly performed in this data structure. Update the code provided in this section to support these two queries efficiently: `int numDisjointSets()` that returns the number of disjoint sets currently in the structure and `int sizeOfSet(int i)` that returns the size of set that currently contains item `i`.

Exercise 2.4.2.2*: Given 8 disjoint sets: {0, 1, 2, ..., 7}, please create a sequence of `unionSet(i, j)` operations to create a tree with rank = 3! Is this possible for rank = 4?

Profiles of Data Structure Inventors

George Boole (1815-1864) was an English mathematician, philosopher, and logician. He is best known to Computer Scientists as the founder of Boolean logic, the foundation of modern digital computers. Boole is regarded as the founder of the field of Computer Science.

Rudolf Bayer (born 1939) has been Professor (emeritus) of Informatics at the Technical University of Munich. He invented the Red-Black (RB) tree used in the C++ STL `map`/`set`.

Georgii Adelson-Velskii (born 1922) is a Soviet mathematician and computer scientist. Along with Evgenii Mikhailovich Landis, he invented the AVL tree in 1962.

Evgenii Mikhailovich Landis (1921-1997) was a Soviet mathematician. The name of the AVL tree is an abbreviation of the two inventors: Adelson-Velskii and Landis himself.

2.4.3   Segment Tree

In this subsection, we will discuss a data structure which can efficiently answer *dynamic*[25] range queries. One such range query is the problem of finding the index of the minimum element in an array within range `[i..j]`. This is more commonly known as the Range Minimum Query (RMQ) problem. For example, given an array `A` of size $n = 7$ below, `RMQ(1, 3) = 2`, as the index 2 contains the minimum element among `A[1]`, `A[2]`, and `A[3]`. To check your understanding of RMQ, verify that in the array A below, `RMQ(3, 4) = 4`, `RMQ(0, 0) = 0`, `RMQ(0, 1) = 1`, and `RMQ(0, 6) = 5`. For the next few paragraphs, assume that array A is the same.

Array	Values	18	17	13	19	15	11	20
A	Indices	0	1	2	3	4	5	6

There are several ways to implement the RMQ. One trivial algorithm is to simply iterate the array from index `i` to `j` and report the index with the minimum value, but this will run in $O(n)$ time per query. When n is large and there are many queries, such an algorithm may be infeasible.

In this section, we answer the dynamic RMQ problem with a Segment Tree, which is another way to arrange data in a binary tree. There are several ways to implement the Segment Tree. Our implementation uses the same concept as the 1-based compact array in the binary heap where we use vi (our shortcut for `vector<int>`) st to represent the binary tree. Index 1 (skipping index 0) is the root and the left and right children of index p are index $2 \times p$ and $(2 \times p) + 1$ respectively (also see Binary Heap discussion in Section 2.3). The value of `st[p]` is the RMQ value of the segment associated with index p.

The root of segment tree represents segment `[0, n-1]`. For each segment `[L, R]` stored in index p where `L != R`, the segment will be split into `[L, (L+R)/2]` and `[(L+R)/2+1, R]` in a left and right vertices. The left sub-segment and right sub-segment will be stored in index $2 \times p$ and $(2 \times p) + 1$ respectively. When `L = R`, it is clear that `st[p] = L` (or `R`). Otherwise, we will recursively build the segment tree, comparing the minimum value of the left and the right sub-segments and updating the `st[p]` of the segment. This process is implemented in the `build` routine below. This `build` routine creates up to $O(1 + 2 + 4 + 8 + \ldots + 2^{\log_2 n}) = O(2n)$ (smaller) segments and therefore runs in $O(n)$. However, as we use simple 1-based compact array indexing, we need `st` to be at least of size $2 * 2^{\lfloor (\log_2(n)) \rfloor + 1}$. In our implementation, we simply use a loose upper bound of space complexity $O(4n) = O(n)$. For array `A` above, the corresponding segment tree is shown in Figure 2.8 and 2.9.

With the segment tree ready, answering an RMQ can be done in $O(\log n)$. The answer for `RMQ(i, i)` is trivial—simply return `i` itself. However, for the general case `RMQ(i, j)`, further checks are needed. Let `p1 = RMQ(i, (i+j)/2)` and `p2 = RMQ((i+j)/2 + 1, j)`. Then `RMQ(i, j)` is `p1` if `A[p1]` $\leq$ `A[p2]` or `p2` otherwise. This process is implemented in the `rmq` routine below.

Take for example the query `RMQ(1, 3)`. The process in Figure 2.8 is as follows: Start from the root (index 1) which represents segment `[0, 6]`. We cannot use the stored minimum value of segment `[0, 6] = st[1] = 5` as the answer for `RMQ(1, 3)` since it is the minimum value over a larger[26] segment than the desired `[1, 3]`. From the root, we only have to go to the left subtree as the root of the right subtree represents segment `[4, 6]` which is outside[27] the desired range in `RMQ(1, 3)`.

---

[25]For dynamic problems, we need to frequently *update* and query the data. This makes pre-processing techniques useless.

[26]Segment `[L, R]` is said to be larger than query range `[i, j]` if `[L, R]` is not outside the query range and not inside query range (see the other footnotes).

[27]Segment `[L, R]` is said to be outside query range `[i, j]` if `i > R || j < L`.
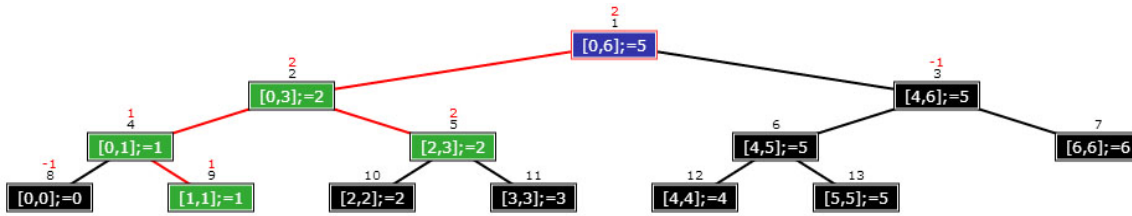
Figure 2.8: Segment Tree of Array A = {18, 17, 13, 19, 15, 11, 20} and `RMQ(1, 3)`

We are now at the root of the left subtree (index 2) that represents segment `[0, 3]`. This segment `[0, 3]` is still larger than the desired `RMQ(1, 3)`. In fact, `RMQ(1, 3)` intersects *both* the left sub-segment `[0, 1]` (index 4) and the right sub-segment `[2, 3]` (index 5) of segment `[0, 3]`, so we have to explore *both* subtrees (sub-segments).

The left segment `[0, 1]` (index 4) of `[0, 3]` (index 2) is not yet inside the `RMQ(1, 3)`, so another split is necessary. From segment `[0, 1]` (index 4), we move right to segment `[1, 1]` (index 9), which is now inside[28] `[1, 3]`. At this point, we know that `RMQ(1, 1)` = `st[9]` = 1 and we can return this value to the caller. The right segment `[2, 3]` (index 5) of `[0, 3]` (index 2) is inside the required `[1, 3]`. From the stored value inside this vertex, we know that `RMQ(2, 3)` = `st[5]` = 2. We do *not* need to traverse further down.

Now, back in the call to segment `[0, 3]` (index 2), we now have `p1` = `RMQ(1, 1)` = 1 and `p2` = `RMQ(2, 3)` = 2. Because `A[p1]` > `A[p2]` since `A[1]` = 17 and `A[2]` = 13, we now have `RMQ(1, 3)` = `p2` = 2. This is the final answer.
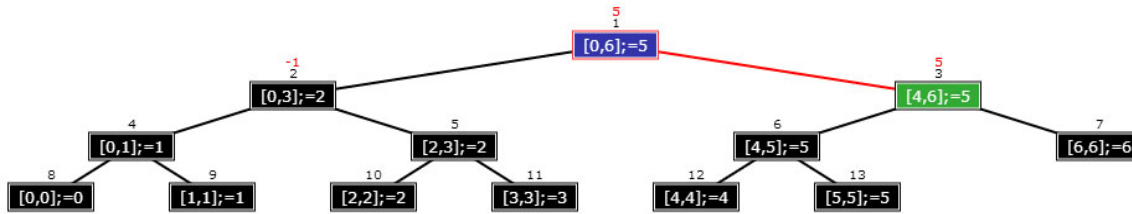
Figure 2.9: Segment Tree of Array A = {18, 17, 13, 19, 15, 11, 20} and `RMQ(4, 6)`

Now let's take a look at another example: `RMQ(4, 6)`. The execution in Figure 2.9 is as follows: We again start from the root segment `[0, 6]` (index 1). Since it is larger than the `RMQ(4, 6)`, we move right to segment `[4, 6]` (index 3) as segment `[0, 3]` (index 2) is outside. Since this segment exactly represents `RMQ(4, 6)`, we simply return the index of minimum element that is stored in this vertex, which is 5. Thus `RMQ(4, 6)` = `st[3]` = 5.

This data structure allows us to avoid traversing the unnecessary parts of the tree! In the worst case, we have *two* root-to-leaf paths which is just $O(2 \times \log(2n)) = O(\log n)$. Example: In `RMQ(3, 4)` = 4, we have one root-to-leaf path from `[0, 6]` to `[3, 3]` (index $1 \to 2 \to 5 \to 11$) and another root-to-leaf path from `[0, 6]` to `[4, 4]` (index $1 \to 3 \to 6 \to 12$).

If the array `A` is static (i.e. unchanged after it is instantiated), then using a Segment Tree to solve the RMQ problem is *overkill* as there exists a Dynamic Programming (DP) solution that requires $O(n \log n)$ one-time pre-processing and allows for $O(1)$ per RMQ. This DP solution will be discussed later in Section 9.33.

Segment Tree is useful if the underlying array is frequently updated (dynamic). For example, if `A[5]` is now changed from 11 to 99, then we just need to update the vertices along the leaf to root path in $O(\log n)$. See path: `[5, 5]` (index 13, `st[13]` is unchanged) $\to$ `[4, 5]` (index 6, `st[6]` = 4 now) $\to$ `[4, 6]` (index 3, `st[3]` = 4 now) $\to$ `[0, 6]` (index

---

[28]Segment `[L, R]` is said to be inside query range `[i, j]` if `L >= i && R <= j`.

1, st[1] = 2 now) in Figure 2.10. For comparison, the DP solution presented in Section 9.33 requires another $O(n \log n)$ pre-processing to update the structure and is ineffective for such dynamic updates.
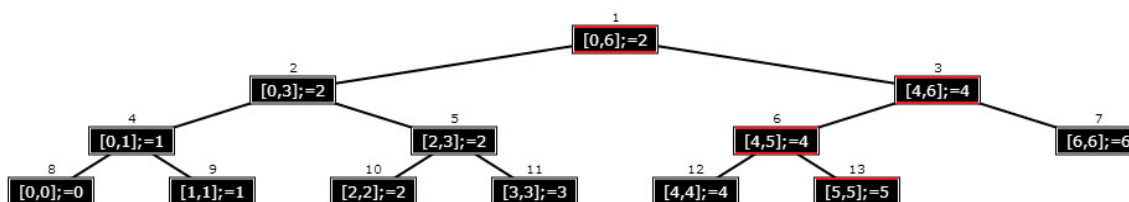
Figure 2.10: Updating Array A to {18, 17, 13, 19, 15, **99**, 20}

Our Segment Tree implementation is shown below. The code shown here supports only *static* RMQs (*dynamic* updates are left as an exercise to the reader).

```
class SegmentTree {            // the segment tree is stored like a heap array
private: vi st, A;             // recall that vi is: typedef vector<int> vi;
  int n;
  int left (int p) { return p << 1; }      // same as binary heap operations
  int right(int p) { return (p << 1) + 1; }

  void build(int p, int L, int R) {                              // O(n)
    if (L == R)                              // as L == R, either one is fine
      st[p] = L;                                        // store the index
    else {                               // recursively compute the values
      build(left(p) , L                , (L + R) / 2);
      build(right(p), (L + R) / 2 + 1, R            );
      int p1 = st[left(p)], p2 = st[right(p)];
      st[p] = (A[p1] <= A[p2]) ? p1 : p2;
  } }

  int rmq(int p, int L, int R, int i, int j) {                // O(log n)
    if (i >  R || j <  L) return -1; // current segment outside query range
    if (L >= i && R <= j) return st[p];            // inside query range

     // compute the min position in the left and right part of the interval
    int p1 = rmq(left(p) , L                , (L+R) / 2, i, j);
    int p2 = rmq(right(p), (L+R) / 2 + 1, R            , i, j);

    if (p1 == -1) return p2;   // if we try to access segment outside query
    if (p2 == -1) return p1;                          // same as above
    return (A[p1] <= A[p2]) ? p1 : p2;                // as in build routine
  }

public:
  SegmentTree(const vi &_A) {
    A = _A; n = (int)A.size();                  // copy content for local usage
    st.assign(4 * n, 0);                 // create large enough vector of zeroes
    build(1, 0, n - 1);                              // recursive build
  }
```

```
    int rmq(int i, int j) { return rmq(1, 0, n - 1, i, j); }   // overloading
};

int main() {
  int arr[] = { 18, 17, 13, 19, 15, 11, 20 };         // the original array
  vi A(arr, arr + 7);
  SegmentTree st(A);
  printf("RMQ(1, 3) = %d\n", st.rmq(1, 3));            // answer = index 2
  printf("RMQ(4, 6) = %d\n", st.rmq(4, 6));            // answer = index 5
} // return 0;
```

Visualization: `www.comp.nus.edu.sg/~stevenha/visualization/segmenttree.html`

Source code: `ch2_09_segmenttree_ds.cpp/java`

Exercise 2.4.3.1*: Draw the Segment Tree corresponding to array `A` = {10, 2, 47, 3, 7, 9, 1, 98, 21}. Answer RMQ(1, 7) and RMQ(3, 8)! Hint: Use the Segment Tree visualization tool shown above.

Exercise 2.4.3.2*: In this section, we have seen how Segment Trees can be used to answer Range Minimum Queries (RMQs). Segment Trees can also be used to answer dynamic Range Sum Queries (`RSQ(i, j)`), i.e. a sum from `A[i] + A[i + 1] + ...+ A[j]`. Modify the given Segment Tree code above to deal with RSQ.

Exercise 2.4.3.3: Using a similar Segment Tree to **Exercise 2.4.3.1** above, answer the queries `RSQ(1, 7)` and `RSQ(3, 8)`. Is this a good approach to solve the problem if array `A` is never changed? (also see Section 3.5.2).

Exercise 2.4.3.4*: The Segment Tree code shown above lacks the (point) `update` operation as discussed in the body text. Add the $O(\log n)$ `update` function to update the value of a certain index (point) in array `A` and simultaneously update the corresponding Segment Tree!

Exercise 2.4.3.5*: The (point) update operation shown in the body text only changes the value of a certain index in array `A`. What if we delete existing elements of array `A` or insert a new elements into array `A`? Can you explain what will happen with the given Segment Tree code and what you should do to address it?

Exercise 2.4.3.6*: There is also one more important Segment Tree operation that has not yet been discussed, the *range* update operation. Suppose a certain subarray of `A` is updated to a certain common value. Can we update the Segment Tree efficiently? Study and solve UVa 11402 - Ahoy Pirates—a problem that requires range updates.

2.4.4 Binary Indexed (Fenwick) Tree

Fenwick Tree—also known as **Binary Indexed Tree** (BIT)—were invented by *Peter M. Fenwick* in 1994 [18]. In this book, we will use the term Fenwick Tree as opposed to BIT in order to differentiate with the standard *bit manipulations*. The Fenwick Tree is a useful data structure for implementing *dynamic cumulative frequency tables*. Suppose we have[29] test scores of `m = 11` students `f = {2,4,5,5,6,6,6,7,7,8,9}` where the test scores are *integer values* ranging from `[1..10]`. Table 2.1 shows the frequency of each individual test score $\in$ `[1..10]` and the cumulative frequency of test scores ranging from `[1..i]` denoted by `cf[i]`—that is, the sum of the frequencies of test scores `1, 2, ..., i`.

Index/ Score	Frequency f	Cumulative Frequency cf	Short Comment
0	-	-	Index 0 is ignored (as the sentinel value).
1	0	0	cf[1] = f[1] = 0.
2	1	1	cf[2] = f[1] + f[2] = 0 + 1 = 1.
3	0	1	cf[3] = f[1] + f[2] + f[3] = 0 + 1 + 0 = 1.
4	1	2	cf[4] = cf[3] + f[4] = 1 + 1 = 2.
5	2	4	cf[5] = cf[4] + f[5] = 2 + 2 = 4.
6	3	7	cf[6] = cf[5] + f[6] = 4 + 3 = 7.
7	2	9	cf[7] = cf[6] + f[7] = 7 + 2 = 9.
8	1	10	cf[8] = cf[7] + f[8] = 9 + 1 = 10.
9	1	11	cf[9] = cf[8] + f[9] = 10 + 1 = 11.
10	0	11	cf[10] = cf[9] + f[10] = 11 + 0 = 11.

Table 2.1: Example of a Cumulative Frequency Table

The cumulative frequency table can also be used as a solution to the Range Sum Query (RSQ) problem mentioned in **Exercise 2.4.3.2***. It stores `RSQ(1, i)` $\forall i \in$ `[1..n]` where `n` is the largest integer index/score[30]. In the example above, we have `n = 10`, `RSQ(1, 1) = 0`, `RSQ(1, 2) = 1`, ..., `RSQ(1, 6) = 7`, ..., `RSQ(1, 8) = 10`, ..., and `RSQ(1, 10) = 11`. We can then obtain the answer to the RSQ for an arbitrary range `RSQ(i, j)` when $i \neq 1$ by subtracting `RSQ(1, j) - RSQ(1, i - 1)`. For example, `RSQ(4, 6) = RSQ(1, 6) - RSQ(1, 3) = 7 - 1 = 6`.

If the frequencies are *static*, then the cumulative frequency table as in Table 2.1 can be computed efficiently with a simple $O(n)$ loop. First, set `cf[1] = f[1]`. Then, for `i` $\in$ `[2..n]`, compute `cf[i] = cf[i - 1] + f[i]`. This will be discussed further in Section 3.5.2. However, when the frequencies are frequently updated (increased or decreased) and the RSQs are frequently asked afterwards, it is better to use a *dynamic* data structure.

Instead of using a Segment Tree to implement a *dynamic* cumulative frequency table, we can implement the *far simpler* Fenwick Tree instead (compare the source code for both implementations, provided in this section and in the previous Section 2.4.3). This is perhaps one of the reasons why the Fenwick Tree is currently included in the IOI syllabus [20]. Fenwick Tree operations are also extremely efficient as they use fast bit manipulation techniques (see Section 2.2).

In this section, we will use the function `LSOne(i)` (which is actually `(i & (-i))`) extensively, naming it to match its usage in the original paper [18]. In Section 2.2, we have seen that the operation `(i & (-i))` produces the first Least Significant One-bit in `i`.

---

[29]The test scores are shown in sorted order for simplicity, they do not have to be sorted.

[30]Please differentiate `m` = the number of data points and `n` = the largest integer value among the `m` data points. The meaning of `n` in Fenwick Tree is a bit different compared to other data structures in this book.

The Fenwick Tree is typically implemented as an array (we use a `vector` for size flexibility). The Fenwick Tree is a tree that is indexed by the *bits* of its *integer* keys. These integer keys fall within the fixed range `[1..n]`—skipping[31] index 0. In a programming contest environment, `n` can approach $\approx 1M$ so that the Fenwick Tree covers the range `[1..1M]`—large enough for many practical (contest) problems. In Table 2.1 above, the scores `[1..10]` are the integer keys in the corresponding array with size `n = 10` and `m = 11` data points.

Let the name of the Fenwick Tree array be `ft`. Then, the element at index `i` is responsible for elements in the range `[i-LSOne(i)+1..i]` and `ft[i]` stores the cumulative frequency of elements `{i-LSOne(i)+1, i-LSOne(i)+2, i-LSOne(i)+3, .., i}`. In Figure 2.11, the value of `ft[i]` is shown in the circle above index `i` and the range `[i-LSOne(i)+1..i]` is shown as a circle and a bar (if the range spans more than one index) above index `i`. We can see that `ft[4] = 2` is responsible for range `[4-4+1..4] = [1..4]`, `ft[6] = 5` is responsible for range `[6-2+1..6] = [5..6]`, `ft[7] = 2` is responsible for range `[7-1+1..7] = [7..7]`, `ft[8] = 10` is responsible for range `[8-8+1..8] = [1..8]` etc[32].

With such an arrangement, if we want to obtain the cumulative frequency between `[1..b]`, i.e. `rsq(b)`, we simply add `ft[b]`, `ft[b']`, `ft[b'']`, ... until index b^i is 0. This sequence of indices is obtained via subtracting the Least Significant One-bit via the bit manipulation expression: `b' = b - LSOne(b)`. Iteration of this bit manipulation effectively *strips off* the least significant one-bit of `b` at each step. As an integer `b` only has $O(\log b)$ bits, `rsq(b)` runs in $O(\log n)$ time when `b = n`. In Figure 2.11, `rsq(6) = ft[6] + ft[4] = 5 + 2 = 7`. Notice that indices 4 and 6 are responsible for range `[1..4]` and `[5..6]`, respectively. By combining them, we account for the entire range of `[1..6]`. The indices 6, 4, and 0 are related in their binary form: `b = `6_{10}` = (1`$\underline{1}$`0)`$_2$ can be transformed to `b' = `4_{10}` = (`$\underline{1}$`00)`$_2$ and subsequently to `b'' = `0_{10}` = (000)`$_2$.
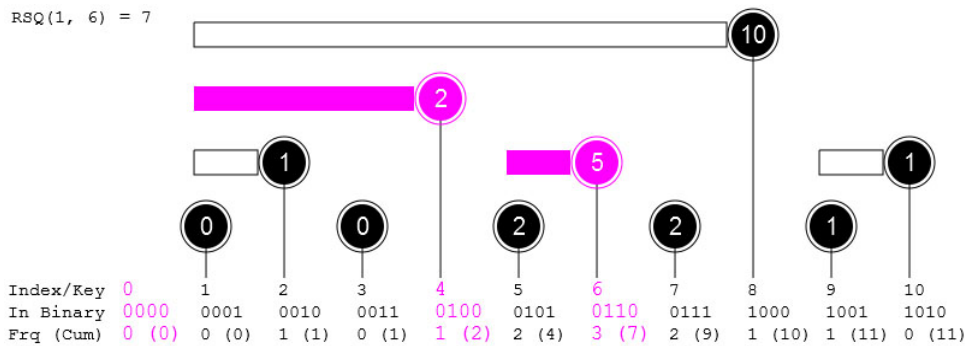
Figure 2.11: Example of `rsq(6)`

With `rsq(b)` available, obtaining the cumulative frequency between two indices `[a..b]` where `a != 1` is simple, just evaluate `rsq(a, b) = rsq(b) - rsq(a - 1)`. For example, if we want to compute `rsq(4, 6)`, we can simply return `rsq(6) - rsq(3) = (5+2) - (0+1) = 7 - 1 = 6`. Again, this operation runs in $O(2 \times \log b) \approx O(\log n)$ time when `b = n`. Figure 2.12 displays the value of `rsq(3)`.

When updating the value of the element at index `k` by adjusting its value by `v` (note that `v` can be either positive or negative), i.e. calling `adjust(k, v)`, we have to update `ft[k]`, `ft[k']`, `ft[k'']`, ... until index k^i exceeds n. This sequence of indices are obtained

---

[31]We have chosen to follow the original implementation by [18] that ignores index 0 to facilitate an easier understanding of the bit manipulation operations of Fenwick Tree. Note that index 0 has no bit turned on. Thus, the operation `i +/- LSOne(i)` simply returns `i` when `i = 0`. Index 0 is also used as the terminating condition in the `rsq` function.

[32]In this book, we will not detail why this arrangement works and will instead show that it allows for efficient $O(\log n)$ update and RSQ operations. Interested readers are advised to read [18].
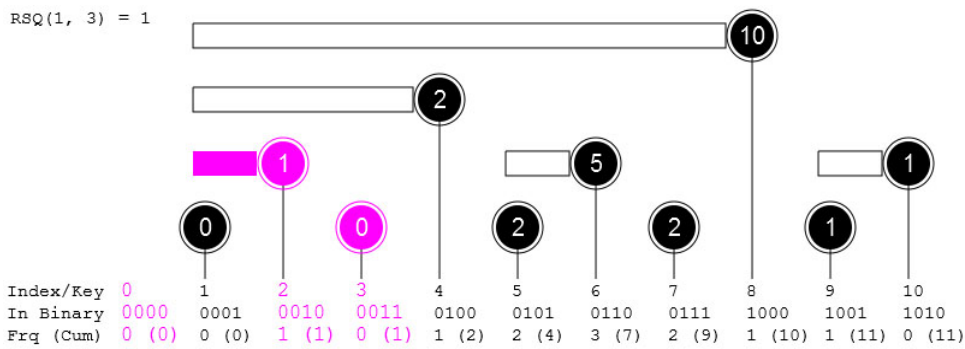
Figure 2.12: Example of `rsq(3)`

via this similar iterative bit manipulation expression: `k' = k + LSOne(k)`. Starting from any integer `k`, the operation `adjust(k, v)` will take at most $O(\log n)$ steps until `k > n`. In Figure 2.13, `adjust(5, 1)` will affect (add +1 to) `ft[k]` at indices `k = `5_{10}` = `$(101)_2$`, k' = `$(101)_2$` + `$(00\underline{1})_2$` = `$(110)_2$` = `6_{10}, and `k'' = `$(110)_2$` + `$(0\underline{1}0)_2$` = `$(1000)_2$` = `8_{10} via the expression given above. Notice that if you project a line upwards from index 5 in Figure 2.13, you will see that the line indeed *intersects* the ranges under the responsibility of index 5, index 6, and index 8.
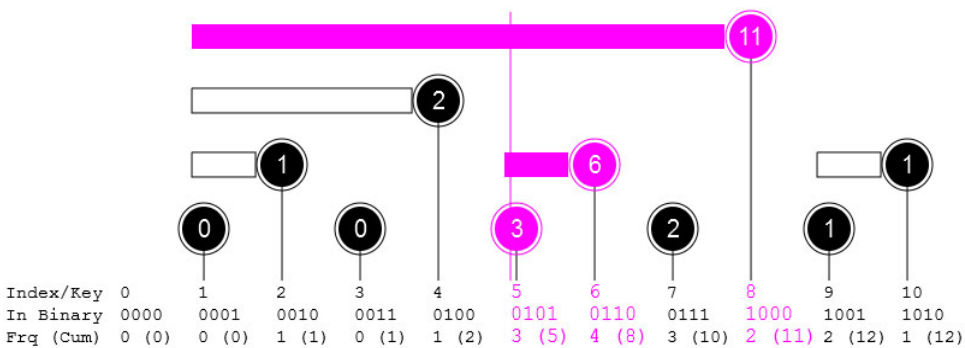
Figure 2.13: Example of `adjust(5, 1)`

In summary, Fenwick Tree supports both RSQ and update operations in just $O(n)$ space and $O(\log n)$ time given a set of m integer keys that ranges from `[1..n]`. This makes Fenwick Tree an ideal data structure for solving *dynamic* RSQ problems on with discrete arrays (the *static* RSQ problem can be solved with simple $O(n)$ pre-processing and $O(1)$ per query as shown earlier). Our *short* C++ implementation of a basic Fenwick Tree is shown below.

```
class FenwickTree {
private: vi ft;                    // recall that vi is: typedef vector<int> vi;
public: FenwickTree(int n) { ft.assign(n + 1, 0); }    // init n + 1 zeroes
  int rsq(int b) {                              // returns RSQ(1, b)
    int sum = 0; for (; b; b -= LSOne(b)) sum += ft[b];
    return sum; }                        // note: LSOne(S) (S & (-S))
  int rsq(int a, int b) {                         // returns RSQ(a, b)
    return rsq(b) - (a == 1 ? 0 : rsq(a - 1)); }
  // adjusts value of the k-th element by v (v can be +ve/inc or -ve/dec)
  void adjust(int k, int v) {                   // note: n = ft.size() - 1
    for (; k < (int)ft.size(); k += LSOne(k)) ft[k] += v; }
};
```

```
int main() {
  int f[] = { 2,4,5,5,6,6,6,7,7,8,9 };                    // m = 11 scores
  FenwickTree ft(10);           // declare a Fenwick Tree for range [1..10]
     // insert these scores manually one by one into an empty Fenwick Tree
  for (int i = 0; i < 11; i++) ft.adjust(f[i], 1);    // this is O(k log n)
  printf("%d\n", ft.rsq(1, 1));   // 0 => ft[1] = 0
  printf("%d\n", ft.rsq(1, 2));   // 1 => ft[2] = 1
  printf("%d\n", ft.rsq(1, 6));   // 7 => ft[6] + ft[4] = 5 + 2 = 7
  printf("%d\n", ft.rsq(1, 10));  // 11 => ft[10] + ft[8] = 1 + 10 = 11
  printf("%d\n", ft.rsq(3, 6));   // 6 => rsq(1, 6) - rsq(1, 2) = 7 - 1
  ft.adjust(5, 2);                // update demo
  printf("%d\n", ft.rsq(1, 10)); // now 13
} // return 0;
```

Visualization: `www.comp.nus.edu.sg/~stevenha/visualization/bit.html`

Source code: `ch2_10_fenwicktree_ds.cpp/java`

Exercise 2.4.4.1: Just a simple exercise of the two basic bit-manipulation operations used in the Fenwick Tree: What are the values of 90 - `LSOne(90)` and 90 + `LSOne(90)`?

Exercise 2.4.4.2: What if the problem that you want to solve includes an element at integer key 0? Recall that the standard integer key range in our library code is is `[1..n]` and that this implementation cannot use index 0 since it is used as the terminating condition of `rsq`.

Exercise 2.4.4.3: What if the problem that you want to solve uses non-integer keys? For example, what if the test scores shown in Table 2.1 above are `f = {5.5, 7.5, 8.0, 10.0}` (i.e. allowing either a 0 or a 5 after the decimal place)? What if the test scores are `f = {5.53, 7.57, 8.10, 9.91}` (i.e. allowing for two digits after the decimal point)?

Exercise 2.4.4.4: The Fenwick Tree supports an additional operation that we have decided to leave as an exercise to the reader: Find the smallest index with a given cumulative frequency. For example, we may need to determine the minimum index/score i in Table 2.1 such that there are at least 7 students covered in the range `[1..i]` (index/score 6 in this case). Implement this feature.

Exercise 2.4.4.5*: Solve this dynamic RSQ problem: UVa 12086 - Potentiometers using *both* a Segment Tree and Fenwick Tree. Which solution is easier to produce in this case? Also see Table 2.2 for a comparison between these two data structures.

Exercise 2.4.4.6*: Extend the 1D Fenwick Tree to 2D!

Exercise 2.4.4.7*: Fenwick Trees are normally used for point update and range (sum) query. Show how to use a Fenwick Tree for *range update* and point queries. For example, given lots of intervals with small ranges (from 1 to at most 1 million) determine the number of intervals encompassing index i.

Profile of Data Structure Inventors

Peter M. Fenwick is a Honorary Associate Professor in the University of Auckland. He invented the Binary Indexed Tree in 1994 [18] as "cumulative frequency tables of arithmetic compression". The BIT has since been included in the IOI syllabus [20] and used in many contest problems for its efficient yet easy to implement data structure.

Feature	Segment Tree	Fenwick Tree
Build Tree from Array	$O(n)$	$O(m \log n)$
Dynamic RMin/MaxQ	OK	Very limited
Dynamic RSQ	OK	OK
Query Complexity	$O(\log n)$	$O(\log n)$
Point Update Complexity	$O(\log n)$	$O(\log n)$
Length of Code	Longer	Shorter

Table 2.2: Comparison Between Segment Tree and Fenwick Tree

Programming exercises that use the data structures discussed and implemented:

- Graph Data Structures Problems

  1. **UVa 00599 - The Forrest for the Trees *** ($v-e =$ number of connected components, keep a `bitset` of size 26 to count the number of vertices that have some edge. Note: Also solvable with Union-Find)
  2. **UVa 10895 - Matrix Transpose *** (transpose adjacency list)
  3. UVa 10928 - My Dear Neighbours (counting out degrees)
  4. UVa 11550 - Demanding Dilemma (graph representation, incidence matrix)
  5. **UVa 11991 - Easy Problem from ... *** (use the idea of an Adj List)
     Also see: More graph problems in Chapter 4

- Union-Find Disjoint Sets

  1. **UVa 00793 - Network Connections *** (trivial; application of disjoint sets)
  2. UVa 01197 - The Suspects (LA 2817, Kaohsiung03, Connected Components)
  3. UVa 10158 - War (advanced usage of disjoint sets with a nice twist; memorize list of enemies)
  4. UVa 10227 - Forests (merge two disjoint sets if they are consistent)
  5. **UVa 10507 - Waking up brain *** (disjoint sets simplifies this problem)
  6. UVa 10583 - Ubiquitous Religions (count disjoint sets after all unions)
  7. UVa 10608 - Friends (find the set with the largest element)
  8. UVa 10685 - Nature (find the set with the largest element)
  9. **UVa 11503 - Virtual Friends *** (maintain set attribute (size) in rep item)
  10. UVa 11690 - Money Matters (check if total money from each member is 0)

- Tree-related Data Structures

  1. UVa 00297 - Quadtrees (simple quadtree problem)
  2. UVa 01232 - SKYLINE (LA 4108, Singapore07, a simple problem if input size is small; but since $n \le 100000$, we have to use a Segment Tree; note that this problem is not about RSQ/RMQ)
  3. **UVa 11235 - Frequent Values *** (range maximum query)
  4. UVa 11297 - Census (Quad Tree with updates or use 2D segment tree)
  5. UVa 11350 - Stern-Brocot Tree (simple tree data structure question)
  6. **UVa 11402 - Ahoy, Pirates *** (segment tree with *lazy* updates)
  7. UVa 12086 - Potentiometers (LA 2191, Dhaka06; pure dynamic range sum query problem; solvable with Fenwick Tree or Segment Tree)
  8. ***UVa 12532 - Interval Product **** (clever usage of Fenwick/Segment Tree)
     Also see: DS as part of the solution of harder problems in Chapter 8

2.5 Solution to Non-Starred Exercises

Exercise 2.2.1*: Sub-question 1: First, sort S in $O(n \log n)$ and then do an $O(n)$ linear scan starting from the second element to check if an integer and the previous integer are the same (also read the solution for **Exercise 1.2.10**, task 4). Sub-question 6: Read the opening paragraph of Chapter 3 and the detailed discussion in Section 9.29. Solutions for the other sub-questions are not shown.

Exercise 2.2.2: The answers (except sub-question 7):

1. $S \, \& \, (N - 1)$

2. $(S \, \& \, (S - 1)) == 0$

3. $S \, \& \, (S - 1)$

4. $S \parallel (S + 1)$

5. $S \, \& \, (S + 1)$

6. $S \parallel (S - 1)$

Exercise 2.3.1: Since the collection is dynamic, we will encounter frequent insertion and deletion queries. An insertion can potentially change the sort order. If we store the information in a static array, we will have to use one $O(n)$ iteration of an insertion sort after each insertion and deletion (to close the gap in the array). This is inefficient!

Exercise 2.3.2:

1. `search(71)`: root $(15) \to 23 \to 71$ (found)
   `search(7)`: root $(15) \to 6 \to 7$ (found)
   `search(22)`: root $(15) \to 23 \to$ empty left subtree (not found).

2. We will eventually have the same BST as in Figure 2.2.

3. To find the min/max element, we can start from root and keep going left/right until we encounter a vertex with no left/right subtrees respectively. That vertex is the answer.

4. We will obtain the sorted output: 4, 5, 6, 7, 15, 23, 50, 71. See Section 4.7.2 if you are not familiar with the inorder tree traversal algorithm.

5. `successor(23)`: Find the minimum element of the subtree rooted at the right of 23, which is the subtree rooted at 71. The answer is 50.
   `successor(7)`: 7 has no right subtree, so 7 must be the maximum of a certain subtree. That subtree is the subtree rooted at 6. The parent of 6 is 15 and 6 is the left subtree of 15. By the BST property, 15 must be the successor of 7.
   `successor(71)`: 71 is the largest element and has no successor.
   Note: The algorithm to find the predecessor of a node is similar.

6. `delete(5)`: We simply remove 5, which is a leaf, from the BST
   `delete(71)`: As 71 is an internal vertex with one child, we cannot simply delete 71 as doing so will disconnect the BST into *two* components. We can instead reshuffle the subtree rooted at the parent of 71 (which is 23), causing 23 to has 50 as its right child.

7. `delete(15)`: As 15 is a vertex with two children, we cannot simply delete 15 as doing so will disconnect the BST into *three* components. To deal with this issue, we need to find the successor of 15 (which is 23) and use the successor to replace 15. We then delete the old 23 from the BST (not a problem now). As a note, we can also use predecessor(key) instead of successor(key) during delete(key) for the case when the key has two children.

Exercise 2.3.3*: For Sub-task 1, we run inorder traversal in $O(n)$ and see if the values are sorted. Solutions to other sub-tasks are not shown.

Exercise 2.3.6: The answers:

1. `Insert(26)`: Insert 26 as the left subtree of 3, swap 26 with 3, then swap 26 with 19 and stop. The Max Heap array A now contains {-, 90, 26, 36, 17, 19, 25, 1, 2, 7, 3}.

2. `ExtractMax()`: Swap 90 (maximum element which will be reported after we fix the Max Heap property) with 3 (the current bottom-most right-most leaf/the last item in the Max Heap), swap 3 with 36, swap 3 with 25 and stop. The Max Heap array A now contains {-, 36, 26, 25, 17, 19, 3, 1, 2, 7}.

Exercise 2.3.7: Yes, check that all indices (vertices) satisfy the Max Heap property.

Exercise 2.3.16: Use the C++ STL `set` (or Java `TreeSet`) as it is a balanced BST that supports $O(\log n)$ dynamic insertions and deletions. We can use the inorder traversal to print the data in the BST in sorted order (simply use C++ `iterator`s or Java `Iterator`s).

Exercise 2.3.17: Use the C++ STL `map` (Java `TreeMap`) and a counter variable. A hash table is also a possible solution but not necessary for programming contests. This trick is quite frequently used in various (contest) problems. Example usage:

```
char str[1000];
map<string, int> mapper;
int i, idx;
for (i = idx = 0; i < M; i++) {                    // idx starts from 0
  scanf("%s", &str);
  if (mapper.find(str) == mapper.end())   // if this is the first encounter
  // alternatively, we can also test if mapper.count(str) is greater than 0
    mapper[str] = idx++;        // give str the current idx and increase idx
}
```

Exercise 2.4.1.3: The graph is undirected.

Exercise 2.4.1.4*: Subtask 1: To count the number of vertices of a graph: Adjacency Matrix/Adjacency List → report the number of rows; Edge List → count the number of distinct vertices in all edges. To count the number of edges of a graph: Adjacency Matrix → sum the number of non-zero entries in every row; Adjacency List → sum the length of all the lists; Edge List → simply report the number of rows. Solutions to other sub-tasks are not shown.

Exercise 2.4.2.1: For `int numDisjointSets()`, use an additional integer counter `numSets`. Initially, during `UnionFind(N)`, set `numSets = N`. Then, during `unionSet(i, j)`, decrease `numSets` by one if `isSameSet(i, j)` returns false. Now, `int numDisjointSets()` can simply return the value of `numSets`.

For `int sizeOfSet(int i)`, we use another `vi setSize(N)` initialized to all ones (each set has only one element). During `unionSet(i, j)`, update the `setSize` array by performing `setSize[find(j)] += setSize[find(i)]` (or the other way around depending on rank) if `isSameSet(i, j)` returns false. Now `int sizeOfSet(int i)` can simply return the value of `setSize[find(i)]`;

These two variants have been implemented in `ch2_08_unionfind_ds.cpp/java`.

Exercise 2.4.3.3: `RSQ(1, 7) = 167` and `RSQ(3, 8) = 139`; No, using a Segment Tree is overkill. There is a simple DP solution that uses an $O(n)$ pre-processing step and takes $O(1)$ time per RSQ (see Section 9.33).

Exercise 2.4.4.1: `90 - LSOne(90) = (1011010)₂ - (10)₂ = (1011000)₂ = 88` and `90 + LSOne(90) = (1011010)₂ + (10)₂ = (1011100)₂ = 92`.

Exercise 2.4.4.2: Simple: shift all indices by one. Index i in the 1-based Fenwick Tree now refers to index $i-1$ in the actual problem.

Exercise 2.4.4.3: Simple: convert the floating point numbers into integers. For the first task, we can multiply every number by two. For the second case, we can multiply all numbers by one hundred.

Exercise 2.4.4.4: The cumulative frequency is sorted, thus we can use a *binary search*. Study the 'binary search for the answer' technique discussed in Section 3.3. The resulting time complexity is $O(\log^2 n)$.

2.6 Chapter Notes

The basic data structures mentioned in Section 2.2-2.3 can be found in almost every data structure and algorithm textbook. References to the C++/Java built-in libraries are available online at: `www.cppreference.com` and `java.sun.com/javase/7/docs/api`. Note that although access to these reference websites are usually provided in programming contests, we suggest that you try to master the syntax of the most common library operations to minimize coding time during actual contests!

One exception is perhaps the *lightweight set of Boolean* (a.k.a bitmask). This *unusual* technique is not commonly taught in data structure and algorithm classes, but it is quite important for competitive programmers as it allows for significant speedups if applied to certain problems. This data structure appears in various places throughout this book, e.g. in some iterative brute force and optimized backtracking routines (Section 3.2.2 and Section 8.2.1), DP TSP (Section 3.5.2), DP with bitmask (Section 8.3.1). All of them use bitmasks instead of `vector<boolean>` or `bitset<size>` due to its efficiency. Interested readers are encouraged to read the book "Hacker's Delight" [69] that discusses bit manipulation in further detail.

Extra references for the data structures mentioned in Section 2.4 are as follows. For Graphs, see [58] and Chapters 22-26 of [7]. For Union-Find Disjoint Sets, see Chapter 21 of [7]. For Segment Trees and other geometric data structures, see [9]. For the Fenwick Tree, see [30]. We remark that all our implementation of data structures discussed in Section 2.4 avoid the usage of pointers. We use either arrays or vectors.

With more experience and by reading the source code we have provided, you can master more tricks in the application of these data structures. Please spend time exploring the source code provided with this book at `sites.google.com/site/stevenhalim/home/material`.

There are few more data structures discussed in this book—string-specific data structures (**Suffix Trie/Tree/Array**) are discussed in Section 6.6. Yet, there are still many other data structures that we cannot cover in this book. If you want to do better in programming contests, please research data structure techniques beyond what we have presented in this book. For example, **AVL Trees**, **Red Black Trees**, or even **Splay Trees** are useful for certain problems that require you to implement and augment (add more data to) balanced BSTs (see Section 9.29). **Interval Tree**s (which are similar to Segment Trees) and **Quad Tree**s (for partitioning 2D space) are useful to know as their underlying concepts may help you to solve certain contest problems.

Notice that many of the efficient data structures discussed in this book exhibit the 'Divide and Conquer' strategy (discussed in Section 3.3).

Statistics	First Edition	Second Edition	Third Edition
Number of Pages	12	18 (+50%)	35 (+94%)
Written Exercises	5	12 (+140%)	14+27*=41 (+242%)
Programming Exercises	43	124 (+188%)	132 (+6%)

The breakdown of the number of programming exercises from each section is shown below:

Section	Title	Appearance	% in Chapter	% in Book
2.2	**Linear DS**	79	60%	5%
2.3	**Non-Linear DS**	30	23%	2%
2.4	Our-own Libraries	23	17%	1%

L-R: Adhiraj, Steven

L-R: Victor, Hung, Dr CJ Hwang, Phong, Duc, Tien

L-R: Hubert, Chuanqi, Zi Chun, Steven, Zhan Xiong, Raymond, Mr Cheong

Chapter 3

Problem Solving Paradigms

If all you have is a hammer, everything looks like a nail
— **Abraham Maslow, 1962**

3.1 Overview and Motivation

In this chapter, we discuss *four* problem solving paradigms commonly used to attack problems in programming contests, namely Complete Search (a.k.a Brute Force), Divide and Conquer, the Greedy approach, and Dynamic Programming. All competitive programmers, including IOI and ICPC contestants, need to master these problem solving paradigms (and more) in order to be able to attack a given problem with the appropriate 'tool'. Hammering *every* problem with Brute Force solutions will not enable anyone to perform well in contests. To illustrate, we discuss four simple tasks below involving an array `A` containing $n \leq 10K$ small integers $\leq 100K$ *(e.g. `A` = {10, 7, 3, 5, 8, 2, 9}, n = 7)* to give an overview of what happens if we attempt every problem with Brute Force as our sole paradigm.

1. Find the largest and the smallest element of `A`. *(10 and 2 for the given example)*.
2. Find the kth smallest element in `A`. *(if $k = 2$, the answer is 3 for the given example)*.
3. Find the largest gap g such that $x, y \in$ `A` and $g = |x - y|$. *(8 for the given example)*.
4. Find the longest increasing subsequence of `A`. *({3, 5, 8, 9} for the given example)*.

The answer for the first task is simple: Try each element of `A` and check if it is the current largest (or smallest) element seen so far. This is an $O(n)$ **Complete Search** solution.

The second task is a little harder. We can use the solution above to find the smallest value and replace it with a large value (e.g. $1M$) to 'delete' it. We can then proceed to find the smallest value again (the second smallest value in the original array) and replace it with $1M$. Repeating this process k times, we will find the kth smallest value. This works, but if $k = \frac{n}{2}$ (the median), this Complete Search solution runs in $O(\frac{n}{2} \times n) = O(n^2)$. Instead, we can sort the array `A` in $O(n \log n)$, returning the answer simply as `A[k-1]`. However, a better solution for a small number of queries is the expected $O(n)$ solution shown in Section 9.29. The $O(n \log n)$ and $O(n)$ solutions above are **Divide and Conquer** solutions.

For the third task, we can similarly consider all possible two integers x and y in `A`, checking if the gap between them is the largest for each pair. This Complete Search approach runs in $O(n^2)$. It works, but is slow and inefficient. We can prove that g can be obtained by finding the difference between the smallest and largest elements of `A`. These two integers can be found with the solution of the first task in $O(n)$. No other combination of two integers in `A` can produce a larger gap. This is a **Greedy** solution.

For the fourth task, trying all $O(2^n)$ possible subsequences to find the longest increasing one is not feasible for all $n \leq 10K$. In Section 3.5.2, we discuss a simple $O(n^2)$ **Dynamic Programming** solution and also the faster $O(n \log k)$ Greedy solution for this task.

Here is some advice for this chapter: Please do not just memorize the solutions for each problem discussed, but instead remember and internalize the thought process and problem solving strategies used. Good problem solving skills are more important than memorized solutions for well-known Computer Science problems when dealing with (often creative and novel) contest problems.

3.2 Complete Search

The Complete Search technique, also known as brute force or recursive backtracking, is a method for solving a problem by traversing the entire (or part of the) search space to obtain the required solution. During the search, we are allowed to prune (that is, choose not to explore) parts of the search space if we have determined that these parts have no possibility of containing the required solution.

In programming contests, a contestant *should* develop a Complete Search solution when there is clearly no other algorithm available (e.g. the task of enumerating *all* permutations of $\{0, 1, 2, \ldots, N-1\}$ clearly requires $O(N!)$ operations) or when better algorithms exist, but are *overkill* as the input size happens to be small (e.g. the problem of answering Range Minimum Queries as in Section 2.4.3 but on static arrays with $N \leq 100$ is solvable with an $O(N)$ loop for each query).

In ICPC, Complete Search should be the first solution considered as it is usually easy to come up with such a solution and to code/debug it. Remember the 'KISS' principle: Keep It Short and Simple. A *bug-free* Complete Search solution should *never* receive the Wrong Answer (WA) response in programming contests as it explores the *entire* search space. However, many programming problems do have better-than-Complete-Search solutions as illustrated in the Section 3.1. Thus a Complete Search solution may receive a Time Limit Exceeded (TLE) verdict. With proper analysis, you can determine the likely outcome (TLE versus AC) before attempting to code anything (Table 1.4 in Section 1.2.3 is a good starting point). If a Complete Search is likely to pass the time limit, then go ahead and implement one. This will then give you more time to work on harder problems in which Complete Search will be too slow.

In IOI, you will usually need better problem solving techniques as Complete Search solutions are usually only rewarded with very small fractions of the total score in the subtask scoring schemes. Nevertheless, Complete Search should be used when you cannot come up with a better solution—it will at least enable you to score some marks.

Sometimes, running Complete Search on *small instances* of a challenging problem can help us to understand its structure through patterns in the output (it is possible to *visualize* the pattern for some problems) that can be exploited to design a faster algorithm. Some combinatorics problems in Section 5.4 can be solved this way. Then, the Complete Search solution can also act as a verifier for *small instances*, providing an additional check for the faster but non-trivial algorithm that you develop.

After reading this section, you may have the impression that Complete Search only works for 'easy problems' and it is usually not the intended solution for 'harder problems'. This is not entirely true. There exist hard problems that are only solvable with creative Complete Search algorithms. We have reserved those problems for Section 8.2.

In the next two sections, we give several *(easier)* examples of this simple yet possibly challenging paradigm. In Section 3.2.1, we give examples that are implemented *iteratively*. In Section 3.2.2, we give examples on solutions that are implemented *recursively* (with backtracking). Finally, in Section 3.2.3, we provide a few tips to give your solution, especially your Complete Search solution, a better chance to pass the required Time Limit.

3.2.1 Iterative Complete Search

Iterative Complete Search (Two Nested Loops: UVa 725 - Division)

Abridged problem statement: Find and display all pairs of 5-digit numbers that collectively use the digits 0 through 9 once each, such that the first number divided by the second is equal to an integer N, where $2 \leq N \leq 79$. That is, `abcde / fghij = N`, where each letter represents a different digit. The first digit of one of the numbers is allowed to be zero, e.g. for `N = 62`, we have `79546 / 01283 = 62`; `94736 / 01528 = 62`.

Quick analysis shows that `fghij` can only range from `01234` to `98765` which is at most $\approx 100K$ possibilities. An even better bound for `fghij` is the range `01234` to `98765 / N`, which has at most $\approx 50K$ possibilities for $N = 2$ and becomes smaller with increasing N. For each attempted `fghij`, we can get `abcde` from `fghij * N` and then check if all 10 digits are different. This is a doubly-nested loop with a time complexity of at most $\approx 50K \times 10 = 500K$ operations per test case. This is small. Thus, an iterative Complete Search is feasible. The main part of the code is shown below (we use a fancy bit manipulation trick shown in Section 2.2 to determine digit uniqueness):

```
for (int fghij = 1234; fghij <= 98765 / N; fghij++) {
  int abcde = fghij * N;   // this way, abcde and fghij are at most 5 digits
  int tmp, used = (fghij < 10000); // if digit f=0, then we have to flag it
  tmp = abcde; while (tmp) { used |= 1 << (tmp % 10); tmp /= 10; }
  tmp = fghij; while (tmp) { used |= 1 << (tmp % 10); tmp /= 10; }
  if (used == (1<<10) - 1)                // if all digits are used, print it
    printf("%0.5d / %0.5d = %d\n", abcde, fghij, N);
}
```

Iterative Complete Search (Many Nested Loops: UVa 441 - Lotto)

In programming contests, problems that are solvable with a *single* loop are usually considered *easy*. Problems which require doubly-nested iterations like UVa 725 - Division above are more challenging but they are not necessarily considered difficult. Competitive programmers must be comfortable writing code with *more than two* nested loops.

Let's take a look at UVa 441 which can be summarized as follows: Given $6 < k < 13$ integers, enumerate all possible subsets of size 6 of these integers in sorted order.

Since the size of the required subset is always 6 and the output has to be sorted lexico-graphically (the input is already sorted), the easiest solution is to use *six* nested loops as shown below. Note that even in the largest test case when $k = 12$, these six nested loops will only produce $_{12}C_6 = 924$ lines of output. This is small.

```
for (int i = 0; i < k; i++)                              // input: k sorted integers
  scanf("%d", &S[i]);
for (int a = 0    ; a < k - 5; a++)                      // six nested loops!
  for (int b = a + 1; b < k - 4; b++)
    for (int c = b + 1; c < k - 3; c++)
      for (int d = c + 1; d < k - 2; d++)
        for (int e = d + 1; e < k - 1; e++)
          for (int f = e + 1; f < k    ; f++)
            printf("%d %d %d %d %d %d\n",S[a],S[b],S[c],S[d],S[e],S[f]);
```

Iterative Complete Search (Loops + Pruning: UVa 11565 - Simple Equations)

Abridged problem statement: Given three integers A, B, and C ($1 \le A, B, C \le 10000$), find three other distinct integers x, y, and z such that $x + y + z = A$, $x \times y \times z = B$, and $x^2 + y^2 + z^2 = C$.

The third equation $x^2 + y^2 + z^2 = C$ is a good starting point. Assuming that C has the largest value of 10000 and y and z are one and two (x, y, z have to be distinct), then the possible range of values for x is $[-100 \dots 100]$. We can use the same reasoning to get a similar range for y and z. We can then write the following triply-nested iterative solution below that requires $201 \times 201 \times 201 \approx 8M$ operations per test case.

```
bool sol = false; int x, y, z;
for (x = -100; x <= 100; x++)
  for (y = -100; y <= 100; y++)
    for (z = -100; z <= 100; z++)
      if (y != x && z != x && z != y &&      // all three must be different
          x + y + z == A && x * y * z == B && x * x + y * y + z * z == C) {
        if (!sol) printf("%d %d %d\n", x, y, z);
        sol = true; }
```

Notice the way a short circuit AND was used to speed up the solution by enforcing a *lightweight* check on whether x, y, and z are all different *before* we check the three formulas. The code shown above already passes the required time limit for this problem, but we can do better. We can also use the second equation $x \times y \times z = B$ and assume that $x = y = z$ to obtain $x \times x \times x < B$ or $x < \sqrt[3]{B}$. The new range of x is $[-22 \dots 22]$. We can also prune the search space by using `if` statements to execute only some of the (inner) loops, or use `break` and/or `continue` statements to stop/skip loops. The code shown below is now much faster than the code shown above (there are a few other optimizations required to solve the extreme version of this problem: UVa 11571 - Simple Equations - Extreme!!):

```
bool sol = false; int x, y, z;
for (x = -22; x <= 22 && !sol; x++) if (x * x <= C)
  for (y = -100; y <= 100 && !sol; y++) if (y != x && x * x + y * y <= C)
    for (z = -100; z <= 100 && !sol; z++)
      if (z != x && z != y &&
          x + y + z == A && x * y * z == B && x * x + y * y + z * z == C) {
        printf("%d %d %d\n", x, y, z);
        sol = true; }
```

Iterative Complete Search (Permutations: UVa 11742 - Social Constraints)

Abridged problem statement: There are $0 < n \le 8$ movie goers. They will sit in the front row in n consecutive open seats. There are $0 \le m \le 20$ seating constraints among them, i.e. movie goer **a** and movie goer **b** must be at most (or at least) **c** seats apart. The question is simple: How many possible seating arrangements are there?

The key part to solve this problem is in realizing that we have to explore **all** permutations (seating arrangements). Once we realize this fact, we can derive this simple $O(m \times n!)$ 'filtering' solution. We set `counter = 0` and then try all possible $n!$ permutations. We increase the `counter` by 1 if the current permutation satisfies all m constraints. When all $n!$ permutations have been examined, we output the final value of `counter`. As the maximum

n is 8 and maximum m is 20, the largest test case will still only require $20 \times 8! = 806400$ operations—a perfectly viable solution.

If you have never written an algorithm to generate all permutations of a set of numbers (see **Exercise 1.2.3**, task 7), you may still be unsure about how to proceed. The simple C++ solution is shown below.

```
#include <algorithm>                    // next_permutation is inside this C++ STL
// the main routine
int i, n = 8, p[8] = {0, 1, 2, 3, 4, 5, 6, 7};      // the first permutation
do {   // try all possible O(n!) permutations, the largest input 8! = 40320
  ...             // check the given social constraint based on 'p' in O(m)
}                            // the overall time complexity is thus O(m * n!)
while (next_permutation(p, p + n));   // this is inside C++ STL <algorithm>
```

Iterative Complete Search (Subsets: UVa 12455 - Bars)

Abridged problem statement[1]: Given a list l containing $1 \le n \le 20$ integers, is there a subset of list l that sums to another given integer X?

We can try all 2^n possible subsets of integers, sum the selected integers for each subset in $O(n)$, and see if the sum of the selected integers equals to X. The overall time complexity is thus $O(n \times 2^n)$. For the largest test case when $n = 20$, this is just $20 \times 2^{20} \approx 21M$. This is 'large' but still viable (for reason described below).

If you have never written an algorithm to generate all subsets of a set of numbers (see **Exercise 1.2.3**, task 8), you may still be unsure how to proceed. An easy solution is to use the *binary representation* of integers from 0 to $2^n - 1$ to describe all possible subsets. If you are not familiar with bit manipulation techniques, see Section 2.2. The solution can be written in simple C/C++ shown below (also works in Java). Since bit manipulation operations are (very) fast, the required $21M$ operations for the largest test case are still doable in under a second. Note: A faster implementation is possible (see Section 8.2.1).

```
// the main routine, variable 'i' (the bitmask) has been declared earlier
for (i = 0; i < (1 << n); i++) {                    // for each subset, O(2^n)
  sum = 0;
  for (int j = 0; j < n; j++)                        // check membership, O(n)
    if (i & (1 << j))        // test if bit 'j' is turned on in subset 'i'?
      sum += l[j];                                     // if yes, process 'j'
  if (sum == X) break;                    // the answer is found: bitmask 'i'
}
```

Exercise 3.2.1.1: For the solution of UVa 725, why is it better to iterate through fghij and not through abcde?

Exercise 3.2.1.2: Does a 10! algorithm that permutes abcdefghij work for UVa 725?

Exercise 3.2.1.3*: Java does *not* have a built-in next_permutation function yet. If you are a Java user, write your own recursive backtracking routine to generate all permutations! This is similar to the recursive backtracking for the 8-Queens problem.

Exercise 3.2.1.4*: How would you solve UVa 12455 if $1 \le n \le \underline{30}$ and each integer can be as big as 1000000000? Hint: See Section 8.2.4.

---

[1]This is also known as the 'Subset Sum' problem, see Section 3.5.3.

3.2.2 Recursive Complete Search

Simple Backtracking: UVa 750 - 8 Queens Chess Problem

Abridged problem statement: In chess (with an 8×8 board), it is possible to place eight queens on the board such that no two queens attack each other. Determine *all* such possible arrangements given the position of one of the queens (i.e. coordinate (a, b) must contain a queen). Output the possibilities in lexicographical (sorted) order.

The most naïve solution is to enumerate all combinations of 8 different cells out of the $8 \times 8 = 64$ possible cells in a chess board and see if the 8 queens can be placed at these positions without conflicts. However, there are $_{64}C_8 \approx 4B$ such possibilities—this idea is not even worth trying.

A better but still naïve solution is to realize that each queen can only occupy one column, so we can put exactly one queen in each column. There are only $8^8 \approx 17M$ possibilities now, down from $4B$. This is still a 'borderline'-passing solution for this problem. If we write a Complete Search like this, we are likely to receive the Time Limit Exceeded (TLE) verdict especially if there are multiple test cases. We can still apply the few more easy optimizations described below to further reduce the search space.

We know that no two queens can share the same column *or the same row*. Using this, we can further simplify the original problem to the problem of finding valid *permutations* of 8! row positions. The value of `row[i]` describes the row position of the queen in column `i`. Example: `row = {1, 3, 5, 7, 2, 0, 6, 4}` as in Figure 3.1 is one of the solutions for this problem; `row[0] = 1` implies that the queen in column 0 is placed in row 1, and so on (the index starts from 0 in this example). Modeled this way, the search space goes *down* from $8^8 \approx 17M$ to $8! \approx 40K$. This solution is already fast enough, but we can still do more.

Figure 3.1: 8-Queens

We also know that no two queens can share any of the two diagonal lines. Let queen A be at `(i, j)` and queen B be at `(k, l)`. They attack each other iff `abs(i-k) == abs(j-l)`. This formula means that the vertical and horizontal distances between these two queens are equal, i.e. queen A and B lie on one of each other's two diagonal lines.

A *recursive backtracking* solution places the queens one by one in columns 0 to 7, observing all the constraints above. Finally, if a candidate solution is found, check if at least one of the queens satisfies the input constraints, i.e. `row[b] == a`. This *sub* (i.e. lower than) $O(n!)$ solution will obtain an AC verdict.

We provide our implementation below. If you have never written a recursive backtracking solution before, please scrutinize it and perhaps re-code it in your own coding style.
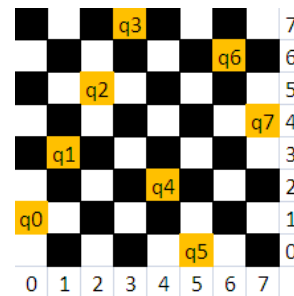
```cpp
#include <cstdlib>                           // we use the int version of 'abs'
#include <cstdio>
#include <cstring>
using namespace std;

int row[8], TC, a, b, lineCounter;           // ok to use global variables

bool place(int r, int c) {
  for (int prev = 0; prev < c; prev++)       // check previously placed queens
    if (row[prev] == r || (abs(row[prev] - r) == abs(prev - c)))
      return false;            // share same row or same diagonal -> infeasible
  return true; }
```

```
void backtrack(int c) {
  if (c == 8 && row[b] == a) {          // candidate sol, (a, b) has 1 queen
    printf("%2d      %d", ++lineCounter, row[0] + 1);
    for (int j = 1; j < 8; j++) printf(" %d", row[j] + 1);
    printf("\n"); }
  for (int r = 0; r < 8; r++)                      // try all possible row
    if (place(r, c)) {           // if can place a queen at this col and row
      row[c] = r; backtrack(c + 1);      // put this queen here and recurse
}    }

int main() {
  scanf("%d", &TC);
  while (TC--) {
    scanf("%d %d", &a, &b); a--; b--;          // switch to 0-based indexing
    memset(row, 0, sizeof row); lineCounter = 0;
    printf("SOLN       COLUMN\n");
    printf(" #       1 2 3 4 5 6 7 8\n\n");
    backtrack(0);               // generate all possible 8! candidate solutions
    if (TC) printf("\n");
} } // return 0;
```

Source code: `ch3_01_UVa750.cpp/java`

More Challenging Backtracking: UVa 11195 - Another n-Queen Problem

Abridged problem statement: Given an $n \times n$ chessboard ($3 < n < \mathbf{15}$) where some of the cells are bad (queens cannot be placed on those bad cells), how many ways can you place n queens in the chessboard so that no two queens attack each other? Note: Bad cells *cannot* be used to block queens' attack.

The recursive backtracking code that we have presented above is *not* fast enough for $n = 14$ and no bad cells, the worst possible test case for this problem. The *sub-$O(n!)$* solution presented earlier is still OK for $n = 8$ but not for $n = 14$. We have to do better.

The major issue with the previous n-queens code is that it is quite slow when checking whether the position of a new queen is valid since we compare the new queen's position with the previous `c-1` queens' positions (see function `bool place(int r, int c)`). It is better to store the same information with three boolean arrays (we use `bitset`s for now):

```
bitset<30> rw, ld, rd;      // for the largest n = 14, we have 27 diagonals
```

Initially all n rows (`rw`), $2 \times n - 1$ left diagonals (`ld`), and $2 \times n - 1$ right diagonals (`rd`) are unused (these three `bitset`s are all set to `false`). When a queen is placed at cell (`r`, `c`), we flag `rw[r] = true` to disallow this row from being used again. Furthermore, all (`a`, `b`) where `abs(r - a) = abs(c - b)` also cannot be used anymore. There are two possibilities after removing the `abs` function: `r - c = a - b` and `r + c = a + b`. Note that `r + c` and `r - c` represent indices for the two diagonal axes. As `r - c` can be negative, we add an *offset* of `n - 1` to both sides of the equation so that `r - c + n - 1 = a - b + n - 1`. If a queen is placed on cell (`r`, `c`), we flag `ld[r - c + n - 1] = true` and `rd[r + c] = true` to disallow these two diagonals from being used again. With these additional data structures and the additional problem-specific constraint in UVa 11195 (`board[r][c]` cannot be a bad cell), we can extend our code to become:

```
void backtrack(int c) {
  if (c == n) { ans++; return; }                               // a solution
  for (int r = 0; r < n; r++)                       // try all possible row
    if (board[r][c] != '*' && !rw[r] && !ld[r - c + n - 1] && !rd[r + c]) {
      rw[r] = ld[r - c + n - 1] = rd[r + c] = true;         // flag off
      backtrack(c + 1);
      rw[r] = ld[r - c + n - 1] = rd[r + c] = false;         // restore
} }
```

Visualization: www.comp.nus.edu.sg/~stevenha/visualization/recursion.html

Exercise 3.2.2.1: The code shown for UVa 750 can be further optimized by pruning the search when 'row[b] != a' *earlier* during the recursion (not only when c == 8). Modify it!

Exercise 3.2.2.2*: Unfortunately, the updated solution presented using bitsets: rw, ld, and rd will still obtain a TLE for UVa 11195 - Another n-Queen Problem. We need to further speed up the solution using bitmask techniques and another way of using the left and right diagonal constraints. This solution will be discussed in Section 8.2.1. For now, use the (non Accepted) idea presented here for UVa 11195 to speed up the code for UVa 750 and two more similar problems: UVa 167 and 11085!

3.2.3 Tips

The biggest gamble in writing a Complete Search solution is whether it will or will not be able to pass the time limit. If the time limit is 10 seconds (online judges do not usually use large time limits for efficient judging) and your program currently runs in ≈ 10 seconds on several (can be more than one) test cases with the largest input size as specified in the problem description, yet your code is still judged to be TLE, you may want to tweak the 'critical code'[2] in your program instead of re-solving the problem with a faster algorithm which may not be easy to design.

Here are some tips that you may want to consider when designing your Complete Search solution for a certain problem to give it a higher chance of passing the Time Limit. Writing a good Complete Search solution is an art in itself.

Tip 1: Filtering versus Generating

Programs that examine lots of (if not all) candidate solutions and choose the ones that are correct (or remove the incorrect ones) are called 'filters', e.g. the naïve 8-queens solver with $_{64}C_8$ and 8^8 time complexity, the iterative solution for UVa 725 and UVa 11742, etc. Usually 'filter' programs are written iteratively.

Programs that gradually build the solutions and immediately prune invalid partial solutions are called 'generators', e.g. the improved recursive 8-queens solver with its *sub-O(n!)* complexity plus diagonal checks. Usually, 'generator' programs are easier to implement when written recursively as it gives us greater flexibility for pruning the search space.

Generally, filters are easier to code but run slower, given that it is usually far more difficult to prune more of the search space iteratively. Do the math (complexity analysis) to see if a filter is good enough or if you need to create a generator.

---

[2]It is said that every program spends most of its time in only about 10% of its code—the critical code.

Tip 2: Prune Infeasible/Inferior Search Space Early

When generating solutions using recursive backtracking (see the tip no 1 above), we may encounter a partial solution that will never lead to a full solution. We can prune the search there and explore other parts of the search space. Example: The diagonal check in the 8-queens solution above. Suppose we have placed a queen at `row[0] = 2`. Placing the next queen at `row[1] = 1` or `row[1] = 3` will cause a diagonal conflict and placing the next queen at `row[1] = 2` will cause a row conflict. Continuing from any of these infeasible partial solutions will never lead to a valid solution. Thus we can prune these partial solutions at this juncture and concentrate only on the other valid positions: `row[1] = {0, 4, 5, 6, 7}`, thus reducing the overall runtime. As a rule of thumb, the earlier you can prune the search space, the better.

In other problems, we may be able to compute the 'potential worth' of a partial (and still valid) solution. If the potential worth is inferior to the worth of the current best found valid solution so far, we can prune the search there.

Tip 3: Utilize Symmetries

Some problems have symmetries and we should try to exploit symmetries to reduce execution time! In the 8-queens problem, there are 92 solutions but there are only 12 unique (or fundamental/canonical) solutions as there are rotational and line symmetries in the problem. You can utilize this fact by only generating the 12 unique solutions and, if needed, generate the whole 92 by rotating and reflecting these 12 unique solutions. Example: `row = {7-1, 7-3, 7-5, 7-7, 7-2, 7-0, 7-6, 7-4} = {6, 4, 2, 0, 5, 7, 1, 3}` is the horizontal reflection of the configuration in Figure 3.1.

However, we have to remark that it is true that sometimes considering symmetries can actually complicate the code. In competitive programming, this is usually not the best way (we want shorter code to minimize bugs). If the gain obtained by dealing with symmetry is not significant in solving the problem, just ignore this tip.

Tip 4: Pre-Computation a.k.a. Pre-Calculation

Sometimes it is helpful to generate tables or other data structures that accelerate the lookup of a result prior to the execution of the program itself. This is called Pre-Computation, in which one trades memory/space for time. However, this technique can rarely be used for recent programming contest problems.

For example, since we know that there are only 92 solutions in the standard 8-queens chess problem, we can create a 2D array `int solution[92][8]` and then fill it with all 92 valid permutations of the 8-queens row positions! That is, we can create a generator program (which takes some time to run) to fill this 2D array `solution`. Afterwards, we can write *another* program to simply and quickly print the correct permutations within the 92 pre-calculated configurations that satisfy the problem constraints.

Tip 5: Try Solving the Problem Backwards

Some contest problems look far easier when they are solved 'backwards' [53] (from a *less obvious* angle) than when they are solved using a frontal attack (from the more obvious angle). Be prepared to attempt unconventional approaches to problems.

This tip is best illustrated using an example: UVa 10360 - Rat Attack: Imagine a 2D array (up to 1024×1024) containing rats. There are $n \leq 20000$ rats spread across the cells. Determine which cell (x, y) should be gas-bombed so that the number of rats killed in

a square box (x-d, y-d) to (x+d, y+d) is maximized. The value d is the power of the gas-bomb ($d \leq 50$), see Figure 3.2.

An immediate solution is to attack this problem in the most obvious fashion possible: bomb each of the 1024^2 cells and select the most effective location. For each bombed cell (x, y), we can perform an $O(d^2)$ scan to count the number of rats killed within the square-bombing radius. For the worst case, when the array has size 1024^2 and $d = 50$, this takes $1024^2 \times 50^2 = 2621M$ operations. TLE[3]!

Another option is to attack this problem backwards: Create an array int killed[1024][1024]. For each rat population at coordinate (x, y), add it to killed[i][j], where $|i - x| \leq d$ and $|j - y| \leq d$. This is because if a bomb was placed at (i, j), the rats at coordinate (x, y) will be killed. This pre-processing takes $O(n \times d^2)$ operations. Then, to determine the most optimal bombing position, we can simply find the coordinate of the highest entry in array killed, which can be done in 1024^2 operations. This approach only requires $20000 \times 50^2 + 1024^2 = 51M$ operations for the worst test case ($n = 20000, d = 50$), ≈ 51 times faster than the frontal attack! This is an AC solution.

● Gas bomb explosion location
◌ Diffusion area, example with d = 1
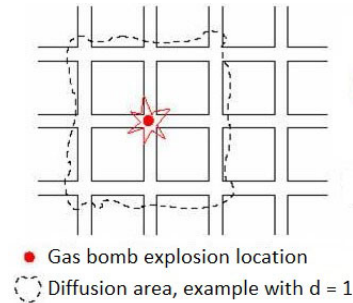
Figure 3.2: UVa 10360 [47]

Tip 6: Optimizing Your Source Code

There are many tricks that you can use to optimize your code. Understanding computer hardware and how it is organized, especially the I/O, memory, and cache behavior, can help you design better code. Some examples (not exhaustive) are shown below:

1. A biased opinion: Use C++ instead of Java. An algorithm implemented using C++ usually runs faster than the one implemented in Java in many online judges, including UVa [47]. Some programming contests give Java users extra time to account for the difference in performance.

2. For C/C++ users, use the faster C-style scanf/printf rather than cin/cout. For Java users, use the faster BufferedReader/BufferedWriter classes as follows:

```
BufferedReader br = new BufferedReader(                  // speedup
  new InputStreamReader(System.in));
// Note: String splitting and/or input parsing is needed afterwards

PrintWriter pr = new PrintWriter(new BufferedWriter(    // speedup
  new OutputStreamWriter(System.out)));
// PrintWriter allows us to use the pr.printf() function
// do not forget to call pr.close() before exiting your Java program
```

3. Use the *expected* $O(n \log n)$ but cache-friendly quicksort in C++ STL algorithm::sort (part of 'introsort') rather than the true $O(n \log n)$ but non cache-friendly heapsort (its root-to-leaf/leaf-to-root operations span a wide range of indices—lots of cache misses).

4. Access a 2D array in a row major fashion (row by row) rather than in a column major fashion—multidimensional arrays are stored in a row-major order in memory.

---

[3]Although 2013 CPU can compute $\approx 100M$ operations in a few seconds, $2621M$ operations will still take too long in a contest environment.

5. Bit manipulation on the built-in integer data types (up to the 64-bit integer) is more efficient than index manipulation in an array of booleans (see bitmask in Section 2.2). If we need more than 64 bits, use the C++ STL `bitset` rather than `vector<bool>` (e.g. for Sieve of Eratosthenes in Section 5.5.1).

6. Use lower level data structures/types at all times if you do not need the extra functionality in the higher level (or larger) ones. For example, use an `array` with a slightly larger size than the maximum size of input instead of using resizable `vector`s. Also, use 32-bit `int`s instead of 64-bit `long long`s as the 32-bit `int` is faster in most 32-bit online judge systems.

7. For Java, use the faster `ArrayList` (and `StringBuilder`) rather than `Vector` (and `StringBuffer`). Java `Vector`s and `StringBuffer`s are *thread safe* but this feature is not needed in competitive programming. Note: In this book, we will stick with `Vector`s to avoid confusing bilingual C++ and Java readers who use both the C++ STL `vector` and Java `Vector`.

8. Declare most data structures (especially the bulky ones, e.g. large arrays) once by placing them in global scope. Allocate enough memory to deal with the largest input of the problem. This way, we do not have to pass the data structures around as function arguments. For problems with multiple test cases, simply clear/reset the contents of the data structure before dealing with each test case.

9. When you have the option to write your code either iteratively or recursively, choose the iterative version. Example: The iterative C++ STL `next_permutation` and iterative subset generation techniques using bitmask shown in Section 3.2.1 are (far) faster than if you write similar routines recursively (mainly due to overheads in function calls).

10. Array access in (nested) loops can be slow. If you have an array `A` and you frequently access the value of `A[i]` (without changing it) in (nested) loops, it may be beneficial to use a local variable `temp = A[i]` and works with `temp` instead.

11. In C/C++, *appropriate* usage of macros or inline functions can reduce runtime.

12. For C++ users: Using C-style character arrays will yield faster execution than when using the C++ STL string. For Java users: Be careful with `String` manipulation as Java `String` objects are immutable. Operations on Java `String`s can thus be very slow. Use Java `StringBuilder` instead.

Browse the Internet or relevant books (e.g. [69]) to find (much) more information on how to speed up your code. Practice this 'code hacking skill' by choosing a harder problem in UVa online judge where the runtime of the best solution is not 0.000s. Submit several variants of your Accepted solution and check the runtime differences. Adopt hacking modification that consistently gives you faster runtime.

Tip 7: Use Better Data Structures & Algorithms :)

No kidding. Using better data structures and algorithms will always outperform any optimizations mentioned in Tips 1-6 above. If you are sure that you have written your fastest Complete Search code, but it is still judged as TLE, abandon the Complete Search approach.

Remarks About Complete Search in Programming Contests

The main source of the 'Complete Search' material in this chapter is the USACO training gateway [48]. We have adopted the name 'Complete Search' rather than 'Brute-Force' (with its negative connotations) as we believe that some Complete Search solutions can be clever and fast. We feel that the term 'clever Brute-Force' is also a little self-contradictory.

If a problem is solvable by Complete Search, it will also be clear when to use the iterative or recursive backtracking approaches. Iterative approaches are used when one can derive the different states *easily* with some formula relative to a certain *counter* and (almost) all states have to be checked, e.g. scanning all the indices of an array, enumerating (almost) all possible subsets of a small set, generating (almost) all permutations, etc. Recursive Backtracking is used when it is hard to derive the different states with a simple index and/or one also wants to (heavily) prune the search space, e.g. the 8-queens chess problem. If the search space of a problem that is solvable with Complete Search is large, then recursive backtracking approaches that allow early pruning of infeasible sections of the search space are usually used. Pruning in iterative Complete Searches is not impossible but usually difficult.

The best way to improve your Complete Search skills is to solve more Complete Search problems. We have provided a list of such problems, separated into several categories below. Please attempt as many as possible, especially those that are highlighted with the **must try *** indicator. Later in Section 3.5, readers will encounter further examples of recursive backtracking, but with the addition of the 'memoization' technique.

Note that we will discuss some *more* advanced search techniques later in Section 8.2, e.g. using bit manipulation in recursive backtracking, harder state-space search, Meet in the Middle, A* Search, Depth Limited Search (DLS), Iterative Deepening Search (IDS), and Iterative Deepening A* (IDA*).

Programming Exercises solvable using Complete Search:

- Iterative (One Loop, Linear Scan)

  1. UVa 00102 - Ecological Bin Packing (just try all 6 possible combinations)
  2. UVa 00256 - Quirksome Squares (brute force, math, pre-calculate-able)
  3. **UVa 00927 - Integer Sequence from ... *** (use sum of arithmetic series)
  4. **UVa 01237 - Expert Enough *** (LA 4142, Jakarta08, input is small)
  5. **UVa 10976 - Fractions Again ? *** (total solutions is asked upfront; therefore do brute force twice)
  6. UVa 11001 - Necklace (brute force math, maximize function)
  7. UVa 11078 - Open Credit System (one linear scan)

- Iterative (Two Nested Loops)

  1. UVa 00105 - The Skyline Problem (height map, sweep left-right)
  2. UVa 00347 - Run, Run, Runaround ... (simulate the process)
  3. UVa 00471 - Magic Numbers (somewhat similar to UVa 725)
  4. UVa 00617 - Nonstop Travel (try all integer speeds from 30 to 60 mph)
  5. UVa 00725 - Division (elaborated in this section)
  6. **UVa 01260 - Sales *** (LA 4843, Daejeon10, check all)
  7. UVa 10041 - Vito's Family (try all possible location of Vito's House)
  8. **UVa 10487 - Closest Sums *** (sort and then do $O(n^2)$ pairings)

9. *UVa 10730 - Antiarithmetic?* (2 nested loops with pruning can pass possibly pass the weaker test cases; note that this brute force solution is too slow for the larger test data generated in the solution of UVa 11129)

10. **UVa 11242 - Tour de France *** (plus sorting)

11. *UVa 12488 - Start Grid* (2 nested loops; simulate overtaking process)

12. *UVa 12583 - Memory Overflow* (2 nested loops; be careful of overcounting)

- Iterative (Three Or More Nested Loops, Easier)

    1. UVa 00154 - Recycling (3 nested loops)
    2. UVa 00188 - Perfect Hash (3 nested loops, until the answer is found)
    3. **UVa 00441 - Lotto *** (6 nested loops)
    4. UVa 00626 - Ecosystem (3 nested loops)
    5. UVa 00703 - Triple Ties: The ... (3 nested loops)
    6. **UVa 00735 - Dart-a-Mania *** (3 nested loops, then count)
    7. **UVa 10102 - The Path in the ... *** (4 nested loops will do, we do not need BFS; get max of minimum Manhattan distance from a '1' to a '3'.)
    8. UVa 10502 - Counting Rectangles (6 nested loops, rectangle, not too hard)
    9. UVa 10662 - The Wedding (3 nested loops)
    10. UVa 10908 - Largest Square (4 nested loops, square, not too hard)
    11. UVa 11059 - Maximum Product (3 nested loops, input is small)
    12. *UVa 11975 - Tele-loto* (3 nested loops, simulate the game as asked)
    13. *UVa 12498 - Ant's Shopping Mall* (3 nested loops)
    14. *UVa 12515 - Movie Police* (3 nested loops)

- Iterative (Three-or-More Nested Loops, Harder)

    1. UVa 00253 - Cube painting (try all, similar problem in UVa 11959)
    2. UVa 00296 - Safebreaker (try all 10000 possible codes, 4 nested loops, use similar solution as 'Master-Mind' game)
    3. UVa 00386 - Perfect Cubes (4 nested loops with pruning)
    4. UVa 10125 - Sumsets (sort; 4 nested loops; plus binary search)
    5. UVa 10177 - (2/3/4)-D Sqr/Rects/... (2/3/4 nested loops, precalculate)
    6. UVa 10360 - Rat Attack (also solvable using 1024^2 DP max sum)
    7. UVa 10365 - Blocks (use 3 nested loops with pruning)
    8. *UVa 10483 - The Sum Equals ...* (2 nested loops for a, b, derive c from a, b; there are 354 answers for range [0.01 .. 255.99]; similar with UVa 11236)
    9. **UVa 10660 - Citizen attention ... *** (7 nested loops, Manhattan distance)
    10. UVa 10973 - Triangle Counting (3 nested loops with pruning)
    11. UVa 11108 - Tautology (5 nested loops, try all $2^5 = 32$ values with pruning)
    12. ***UVa 11236 - Grocery Store **** (3 nested loops for a, b, c; derive d from a, b, c; check if you have 949 lines of output)
    13. UVa 11342 - Three-square (pre-calculate squared values from 0^2 to 224^2, use 3 nested loops to generate the answers; use `map` to avoid duplicates)
    14. *UVa 11548 - Blackboard Bonanza* (4 nested loops, string, pruning)
    15. **UVa 11565 - Simple Equations *** (3 nested loops with pruning)
    16. UVa 11804 - Argentina (5 nested loops)
    17. UVa 11959 - Dice (try all possible dice positions, compare with the 2nd one) Also see Mathematical Simulation in Section 5.2

- Iterative (Fancy Techniques)

  1. UVa 00140 - Bandwidth (max n is just 8, use `next_permutation`; the algorithm inside `next_permutation` is iterative)
  2. *UVa 00234 - Switching Channels* (use `next_permutation`, simulation)
  3. UVa 00435 - Block Voting (only 2^{20} possible coalition combinations)
  4. UVa 00639 - Don't Get Rooked (generate 2^{16} combinations and prune)
  5. **UVa 01047 - Zones *** (LA 3278, WorldFinals Shanghai05, notice that $n \le 20$ so that we can try all possible subsets of towers to be taken; then apply inclusion-exclusion principle to avoid overcounting)
  6. *UVa 01064 - Network* (LA 3808, WorldFinals Tokyo07, permutation of up to 5 messages, simulation, mind the word 'consecutive')
  7. UVa 11205 - The Broken Pedometer (try all 2^{15} bitmask)
  8. UVa 11412 - Dig the Holes (`next_permutation`, find one possibility from 6!)
  9. **UVa 11553 - Grid Game *** (solve by trying all $n!$ permutations; you can also use DP + bitmask, see Section 8.3.1, but it is overkill)
  10. UVa 11742 - Social Constraints (discussed in this section)
  11. *UVa 12249 - Overlapping Scenes* (LA 4994, KualaLumpur10, try all permutations, a bit of string matching)
  12. *UVa 12346 - Water Gate Management* (LA 5723, Phuket11, try all 2^n combinations, pick the best one)
  13. *UVa 12348 - Fun Coloring* (LA 5725, Phuket11, try all 2^n combinations)
  14. *UVa 12406 - Help Dexter* (try all 2^p possible bitmasks, change '0's to '2's)
  15. **UVa 12455 - Bars *** (discussed in this section)

- Recursive Backtracking (Easy)

  1. UVa 00167 - The Sultan Successor (8-queens chess problem)
  2. UVa 00380 - Call Forwarding (simple backtracking, but we have to work with strings, see Section 6.2)
  3. UVa 00539 - The Settlers ... (longest simple path in a *small* general graph)
  4. **UVa 00624 - CD *** (input size is small, backtracking is enough)
  5. UVa 00628 - Passwords (backtracking, follow the rules in description)
  6. UVa 00677 - All Walks of length "n" ... (print all solutions with backtracking)
  7. UVa 00729 - The Hamming Distance ... (generate all possible bit strings)
  8. UVa 00750 - 8 Queens Chess Problem (discussed in this section with sample source code)
  9. UVa 10276 - Hanoi Tower Troubles Again (insert a number one by one)
  10. UVa 10344 - 23 Out of 5 (rearrange the 5 operands and the 3 operators)
  11. UVa 10452 - Marcus, help (at each pos, Indy can go forth/left/right; try all)
  12. **UVa 10576 - Y2K Accounting Bug *** (generate all, prune, take max)
  13. **UVa 11085 - Back to the 8-Queens *** (see UVa 750, pre-calculation)

- Recursive Backtracking (Medium)

  1. UVa 00222 - Budget Travel (looks like a DP problem, but the state cannot be memoized as 'tank' is floating-point; fortunately, the input is not large)
  2. *UVa 00301 - Transportation* (2^{22} with pruning is possible)
  3. UVa 00331 - Mapping the Swaps ($n \le 5...$)
  4. UVa 00487 - Boggle Blitz (use `map` to store the generated words)
  5. **UVa 00524 - Prime Ring Problem *** (also see Section 5.5.1)

6. UVa 00571 - Jugs (solution can be suboptimal, add flag to avoid cycling)

7. **UVa 00574 - Sum It Up *** (print all solutions with backtracking)

8. UVa 00598 - Bundling Newspaper (print all solutions with backtracking)

9. *UVa 00775 - Hamiltonian Cycle* (backtracking suffices because the search space cannot be that big; in a dense graph, it is more likely to have a Hamiltonian cycle, so we can prune early; we do NOT have to find the best one like in TSP problem)

10. *UVa 10001 - Garden of Eden* (the upperbound of 2^{32} is scary but with efficient pruning, we can pass the time limit as the test case is not extreme)

11. *UVa 10063 - Knuth's Permutation* (do as asked)

12. *UVa 10460 - Find the Permuted String* (similar nature with UVa 10063)

13. UVa 10475 - Help the Leaders (generate and prune; try all)

14. **UVa 10503 - The dominoes solitaire *** (max 13 spaces only)

15. *UVa 10506 - Ouroboros* (any valid solution is AC; generate all possible next digit (up to base 10/digit [0..9]); check if it is still a valid Ouroboros sequence)

16. *UVa 10950 - Bad Code* (sort the input; run backtracking; the output should be sorted; only display the first 100 sorted output)

17. UVa 11201 - The Problem with the ... (backtracking involving strings)

18. *UVa 11961 - DNA* (there are at most 4^{10} possible DNA strings; moreover, the mutation power is at most $K \leq 5$ so the search space is much smaller; sort the output and then remove duplicates)

- Recursive Backtracking (Harder)

  1. *UVa 00129 - Krypton Factor* (backtracking, string processing check, a bit of output formatting)

  2. UVa 00165 - Stamps (requires some DP too; can be pre-calculated)

  3. **UVa 00193 - Graph Coloring *** (Max Independent Set, input is small)

  4. UVa 00208 - Firetruck (backtracking with some pruning)

  5. **UVa 00416 - LED Test *** (backtrack, try all)

  6. UVa 00433 - Bank (Not Quite O.C.R.) (similar to UVa 416)

  7. UVa 00565 - Pizza Anyone? (backtracking with lots of pruning)

  8. *UVa 00861 - Little Bishops* (backtracking with pruning as in 8-queens recursive backtracking solution; then pre-calculate the results)

  9. UVa 00868 - Numerical maze (try row 1 to N; 4 ways; some constraints)

  10. **UVa 01262 - Password *** (LA 4845, Daejeon10, sort the columns in the two 6×5 grids first so that we can process common passwords in lexicographic order; backtracking; important: skip two similar passwords)

  11. UVa 10094 - Place the Guards (this problem is like the n-queens chess problem, but must find/use the pattern!)

  12. *UVa 10128 - Queue* (backtracking with pruning; try up to all N! (13!) permutations that satisfy the requirement; then pre-calculate the results)

  13. UVa 10582 - ASCII Labyrinth (simplify complex input first; then backtrack)

  14. *UVa 11090 - Going in Cycle* (minimum mean weight cycle problem; solvable with backtracking with important pruning when current running mean is greater than the best found mean weight cycle cost)

3.3 Divide and Conquer

Divide and Conquer (abbreviated as D&C) is a problem-solving paradigm in which a problem is made *simpler* by 'dividing' it into smaller parts and then conquering each part. The steps:

1. Divide the original problem into *sub*-problems—usually by half or nearly half,

2. Find (sub)-solutions for each of these sub-problems—which are now easier,

3. If needed, combine the sub-solutions to get a complete solution for the main problem.

We have seen examples of the D&C paradigm in the previous sections of this book: Various sorting algorithms (e.g. Quick Sort, Merge Sort, Heap Sort) and Binary Search in Section 2.2 utilize this paradigm. The way data is organized in Binary Search Tree, Heap, Segment Tree, and Fenwick Tree in Section 2.3, 2.4.3, and 2.4.4 also relies upon the D&C paradigm.

3.3.1 Interesting Usages of Binary Search

In this section, we discuss the D&C paradigm in the well-known Binary Search algorithm. We classify Binary Search as a 'Divide' and Conquer algorithm although one reference [40] suggests that it should be actually classified as 'Decrease (by-half)' and Conquer as it does not actually 'combine' the result. We highlight this algorithm because many contestants know it, but not many are aware that it can be used in many other non-obvious ways.

Binary Search: The Ordinary Usage

Recall that the *canonical* usage of Binary Search is searching for an item in a *static sorted array*. We check the middle of the sorted array to determine if it contains what we are looking for. If it is or there are no more items to consider, stop. Otherwise, we can decide whether the answer is to the left or right of the middle element and continue searching. As the size of search space is halved (in a binary fashion) after each check, the complexity of this algorithm is $O(\log n)$. In Section 2.2, we have seen that there are built-in library routines for this algorithm, e.g. the C++ STL `algorithm::lower_bound` (and the Java `Collections.binarySearch`).

This is *not* the only way to use binary search. The pre-requisite for performing a binary search—a *static sorted sequence (array or vector)*—can also be found in other uncommon data structures such as in the root-to-leaf path of a tree (not necessarily binary nor complete) that satisfies the *min heap* property. This variant is discussed below.

Binary Search on Uncommon Data Structures

This original problem is titled 'My Ancestor' and was used in the Thailand ICPC National Contest 2009. Abridged problem description: Given a weighted (family) tree of up to $N \leq 80K$ vertices with a special trait: *Vertex values are increasing from root to leaves.* Find the *ancestor* vertex closest to the root from a starting vertex v that has weight at least P. There are up to $Q \leq 20K$ such *offline* queries. Examine Figure 3.3 (left). If $P = 4$, then the answer is the vertex labeled with 'B' with value 5 as it is the ancestor of vertex v that is closest to root 'A' and has a value of ≥ 4. If $P = 7$, then the answer is 'C', with value 7. If $P \geq 9$, there is no answer.

The naïve solution is to perform a linear $O(N)$ scan per query: Starting from the given vertex v, we move up the (family) tree until we reach the first vertex whose direct parent has value $< P$ or until we reach the root. If this vertex has value $\geq P$ and it is not vertex v
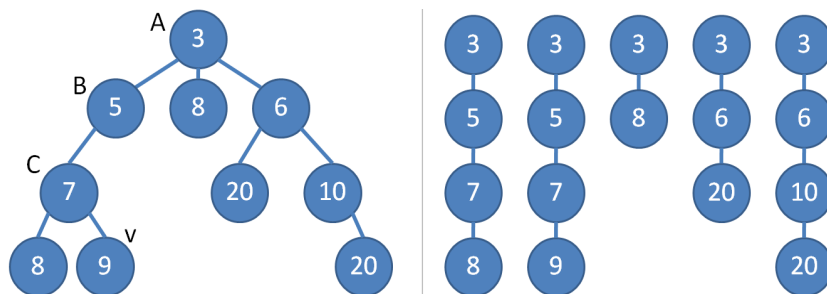
Figure 3.3: My Ancestor (all 5 root-to-leaf paths are sorted)

itself, we have found the solution. As there are Q queries, this approach runs in $O(QN)$ (the input tree can be a sorted linked list, or rope, of length N) and will get a TLE as $N \leq 80K$ and $Q \leq 20K$.

A better solution is to store all the $20K$ queries (we do not have to answer them immediately). Traverse the tree *just once* starting from the root using the $O(N)$ preorder tree traversal algorithm (Section 4.7.2). This preorder tree traversal is slightly modified to remember the partial root-to-current-vertex sequence as it executes. The array is always sorted because the vertices along the root-to-current-vertex path have increasing weights, see Figure 3.3 (right). The preorder tree traversal on the tree shown in Figure 3.3 (left) produces the following partial root-to-current-vertex sorted array: $\{\{3\}, \{3, 5\}, \{3, 5, 7\}$, $\{3, 5, 7, 8\}$, backtrack, $\{3, 5, 7, 9\}$, backtrack, backtrack, backtrack, $\{3, 8\}$, backtrack, $\{3, 6\}, \{3, 6, 20\}$, backtrack, $\{3, 6, 10\}$, and finally $\{3, 6, 10, 20\}$, backtrack, backtrack, backtrack (done)$\}$.

During the preorder traversal, when we land on a queried vertex, we can perform a $O(\log N)$ **binary search** (to be precise: `lower_bound`) on the partial root-to-current-vertex weight array to obtain the ancestor closest to the root with a value of at least P, recording these solutions. Finally, we can perform a simple $O(Q)$ iteration to output the results. The overall time complexity of this approach is $O(Q \log N)$, which is now manageable given the input bounds.

Bisection Method

We have discussed the applications of Binary Searches in finding items in static sorted sequences. However, the binary search **principle**[4] can also be used to find the root of a function that may be difficult to compute directly.

Example: You buy a car with loan and now want to pay the loan in monthly installments of d dollars for m months. Suppose the value of the car is originally v dollars and the bank charges an interest rate of $i\%$ for any unpaid loan at the end of each month. What is the amount of money d that you must pay per month (to 2 digits after the decimal point)?

Suppose $d = 576.19$, $m = 2$, $v = 1000$, and $i = 10\%$. After one month, your debt becomes $1000 \times (1.1) - 576.19 = 523.81$. After two months, your debt becomes $523.81 \times (1.1) - 576.19 \approx 0$. If we are only given $m = 2$, $v = 1000$, and $i = 10\%$, how would we determine that $d = 576.19$? In other words, find the root d such that the debt payment function $f(d, m, v, i) \approx 0$.

An *easy* way to solve this root finding problem is to use the bisection method. We pick a reasonable range as a starting point. We want to fix d within the range `[a..b]` where

---

[4]We use the term 'binary search principle' to refer to the D&C approach of halving the range of possible answers. The 'binary search algorithm' (finding index of an item in a sorted array), the 'bisection method' (finding the root of a function), and 'binary search the answer' (discussed in the next subsection) are all instances of this principle.

$a = 0.01$ as we have to pay at least one cent and $b = (1 + i\%) \times v$ as the earliest we can complete the payment is $m = 1$ if we pay exactly $(1 + i\%) \times v$ dollars after one month. In this example, $b = (1 + 0.1) \times 1000 = 1100.00$ dollars. For the bisection method to work[5], we must ensure that the function values of the two extreme points in the initial Real range `[a..b]`, i.e. $f(a)$ and $f(b)$ have opposite signs (this is true for the computed `a` and `b` above).

a	b	$d = \frac{a+b}{2}$	status: $f(d, m, v, i)$	action
0.01	1100.00	550.005	undershoot by 54.9895	increase d
550.005	1100.00	825.0025	overshoot by 522.50525	decrease d
550.005	825.0025	687.50375	overshoot by 233.757875	decrease d
550.005	687.50375	618.754375	overshoot by 89.384187	decrease d
550.005	618.754375	584.379688	overshoot by 17.197344	decrease d
550.005	584.379688	567.192344	undershoot by 18.896078	increase d
567.192344	584.379688	575.786016	undershoot by 0.849366	increase d
...	...	...	a **few** iterations later ...	...
...	...	576.190476	stop; error is now less than ϵ	answer = 576.19

Table 3.1: Running Bisection Method on the Example Function

Notice that bisection method only requires $O(\log_2((b - a)/\epsilon))$ iterations to get an answer that is good enough (the error is smaller than the threshold error ϵ that we can tolerate). In this example, bisection method only takes $\log_2 1099.99/\epsilon$ tries. Using a small $\epsilon = $ `1e-9`, this yields only ≈ 40 iterations. Even if we use a smaller $\epsilon = $ `1e-15`, we will still only need ≈ 60 tries. Notice that the number of tries is *small*. The bisection method is much more efficient compared to exhaustively evaluating each possible value of $d = $`[0.01..1100.00]`$/\epsilon$ for this example function. Note: The bisection method can be written with a loop that tries the values of $d \approx 40$ to 60 times (see our implementation in the 'binary search the answer' discussion below).

Binary Search the Answer

The abridged version of UVa 11935 - Through the Desert is as follows: Imagine that you are an explorer trying to cross a desert. You use a jeep with a 'large enough' fuel tank – initially full. You encounter a series of events throughout your journey such as 'drive (that consumes fuel)', 'experience gas leak (further reduces the amount of fuel left)', 'encounter gas station (allowing you to refuel to the original capacity of your jeep's fuel tank)', 'encounter mechanic (fixes all leaks)', or 'reach goal (done)'. You need to determine the *smallest possible* fuel tank capacity for your jeep to be able to reach the goal. The answer must be precise to three digits after decimal point.

If we know the jeep's fuel tank capacity, then this problem is just a simulation problem. From the start, we can simulate each event in order and determine if the goal can be reached without running out of fuel. The problem is that we do not know the jeep's fuel tank capacity—this is the value that we are looking for.

From the problem description, we can compute that the range of possible answers is between `[0.000..10000.000]`, with 3 digits of precision. However, there are $10M$ such possibilities. Trying each value sequentially will get us a TLE verdict.

Fortunately, this problem has a property that we can exploit. Suppose that the correct answer is X. Setting your jeep's fuel tank capacity to any value between `[0.000..X-0.001]`

---

[5]Note that the requirements for the bisection method (which uses the binary search principle) are slightly different from the binary search algorithm which needs a sorted array.

will *not* bring your jeep safely to the goal event. On the other hand, setting your jeep fuel tank volume to any value between [X..10000.000] will bring your jeep safely to the goal event, usually with some fuel left. This property allows us to binary search the answer X! We can use the following code to obtain the solution for this problem.

```
#define EPS 1e-9  // this value is adjustable; 1e-9 is usually small enough
bool can(double f) {                  // details of this simulation is omitted
  // return true if the jeep can reach goal state with fuel tank capacity f
  // return false otherwise
}

// inside int main()
  // binary search the answer, then simulate
  double lo = 0.0, hi = 10000.0, mid = 0.0, ans = 0.0;
  while (fabs(hi - lo) > EPS) {        // when the answer is not found yet
    mid = (lo + hi) / 2.0;                          // try the middle value
    if (can(mid)) { ans = mid; hi = mid; } // save the value, then continue
    else          lo = mid;
  }

  printf("%.3lf\n", ans);     // after the loop is over, we have the answer
```

Note that some programmers choose to use a constant number of refinement iterations instead of allowing the number of iterations to vary dynamically to avoid precision errors when testing `fabs(hi - lo) > EPS` and thus being trapped in an infinite loop. The only changes required to implement this approach are shown below. The other parts of the code are the same as above.

```
  double lo = 0.0, hi = 10000.0, mid = 0.0, ans = 0.0;
  for (int i = 0; i < 50; i++) {    // log_2 ((10000.0 - 0.0) / 1e-9) ~= 43
    mid = (lo + hi) / 2.0;     // looping 50 times should be precise enough
    if (can(mid)) { ans = mid; hi = mid; }
    else          lo = mid;
  }
```

Exercise 3.3.1.1: There is an alternative solution for UVa 11935 that does not use 'binary search the answer' technique. Can you spot it?

Exercise 3.3.1.2*: The example shown here involves binary-searching the answer where the answer is a floating point number. Modify the code to solve 'binary search the answer' problems where the answer lies in an *integer range*!

Remarks About Divide and Conquer in Programming Contests

The Divide and Conquer paradigm is usually utilized through popular algorithms that rely on it: Binary Search and its variants, Merge/Quick/Heap Sort, and data structures: Binary Search Tree, Heap, Segment Tree, Fenwick Tree, etc. However—based on our experience, we reckon that the most commonly used form of the Divide and Conquer paradigm in

programming contests is the Binary Search principle. If you want to do well in programming contests, please spend time practicing the various ways to apply it.

Once you are more familiar with the 'Binary Search the Answer' technique discussed in this section, please explore Section 8.4.1 for a few more programming exercises that use this technique with *other algorithm* that we will discuss in the latter parts of this book.

We notice that there are not that many D&C problems outside of our binary search categorization. Most D&C solutions are 'geometry-related' or 'problem specific', and thus cannot be discussed in detail in this book. However, we will encounter some of them in Section 8.4.1 (binary search the answer plus geometry formulas), Section 9.14 (Inversion Index), Section 9.21 (Matrix Power), and Section 9.29 (Selection Problem).

Programming Exercises solvable using Divide and Conquer:

- Binary Search

  1. UVa 00679 - Dropping Balls (binary search; bit manipulation solutions exist)
  2. UVa 00957 - Popes (complete search + binary search: `upper_bound`)
  3. UVa 10077 - The Stern-Brocot ... (binary search)
  4. UVa 10474 - Where is the Marble? (simple: use `sort` and then `lower_bound`)
  5. ***UVa 10567 - Helping Fill Bates **** (store increasing indices of each char of 'S' in 52 vectors; for each query, binary search for the position of the char in the correct vector)
  6. UVa 10611 - Playboy Chimp (binary search)
  7. UVa 10706 - Number Sequence (binary search + some mathematical insights)
  8. UVa 10742 - New Rule in Euphomia (use sieve; binary search)
  9. **UVa 11057 - Exact Sum *** (sort, for price `p[i]`, check if price (`M - p[i]`) exists with binary search)
  10. UVa 11621 - Small Factors (generate numbers with factor 2 and/or 3, `sort`, `upper_bound`)
  11. *UVa 11701 - Cantor* (a kind of ternary search)
  12. UVa 11876 - N + NOD (N) (`[lower|upper]_bound` on sorted sequence N)
  13. ***UVa 12192 - Grapevine **** (the input array has special sorted properties; use `lower_bound` to speed up the search)
  14. Thailand ICPC National Contest 2009 - My Ancestor (author: Felix Halim)

- Bisection Method or Binary Search the Answer

  1. **UVa 10341 - Solve It *** (bisection method discussed in this section; for alternative solutions, see `http://www.algorithmist.com/index.php/UVa_10341`)
  2. **UVa 11413 - Fill the ... *** (binary search the answer + simulation)
  3. UVa 11881 - Internal Rate of Return (bisection method)
  4. UVa 11935 - Through the Desert (binary search the answer + simulation)
  5. ***UVa 12032 - The Monkey ... **** (binary search the answer + simulation)
  6. *UVa 12190 - Electric Bill* (binary search the answer + algebra)
  7. IOI 2010 - Quality of Living (binary search the answer)

     Also see: Divide & Conquer for Geometry Problems (see Section 8.4.1)

- Other Divide & Conquer Problems

  1. ***UVa 00183 - Bit Maps **** (simple exercise of Divide and Conquer)
  2. IOI 2011 - Race (D&C; whether the solution path uses a vertex or not)

     Also see: Data Structures with Divide & Conquer flavor (see Section 2.3)

3.4   Greedy

An algorithm is said to be greedy if it makes the locally optimal choice at each step with the hope of eventually reaching the globally optimal solution. In some cases, greedy works—the solution is short and runs efficiently. For *many* others, however, it does not. As discussed in other typical Computer Science textbooks, e.g. [7, 38], a problem must exhibit these two properties in order for a greedy algorithm to work:

1. It has optimal sub-structures.
   Optimal solution to the problem contains optimal solutions to the sub-problems.

2. It has the greedy property (difficult to prove in time-critical contest environment!).
   If we make a choice that seems like the best at the moment and proceed to solve the remaining subproblem, we reach the optimal solution. We will never have to reconsider our previous choices.

3.4.1   Examples

Coin Change - The Greedy Version

Problem description: Given a target amount V cents and a list of denominations of n coins, i.e. we have `coinValue[i]` (in cents) for coin types `i` $\in$ `[0..n-1]`, what is the minimum number of coins that we must use to represent amount V? Assume that we have an unlimited supply of coins of any type. Example: If $n = 4$, `coinValue = {25, 10, 5, 1}` cents[6], and we want to represent $V = 42$ cents, we can use this Greedy algorithm: Select the largest coin denomination which is not greater than the remaining amount, i.e. 42-<u>25</u> = 17 → 17-<u>10</u> = 7 → 7-<u>5</u> = 2 → 2-<u>1</u> = 1 → 1-<u>1</u> = 0, a total of 5 coins. This is optimal.

The problem above has the two ingredients required for a successful greedy algorithm:

1. It has optimal sub-structures.
   We have seen that in our quest to represent 42 cents, we used 25+10+5+1+1.
   This is an optimal 5-coin solution to the original problem!
   Optimal solutions to sub-problem are contained within the 5-coin solution, i.e.
   a. To represent 17 cents, we can use 10+5+1+1 (part of the solution for 42 cents),
   b. To represent 7 cents, we can use 5+1+1 (also part of the solution for 42 cents), etc

2. It has the greedy property: Given every amount V, we can greedily subtract from it the largest coin denomination which is not greater than this amount V. It can be proven (not shown here for brevity) that using any other strategies will not lead to an optimal solution, at least for this set of coin denominations.

However, this greedy algorithm does *not* work for *all* sets of coin denominations. Take for example {4, 3, 1} cents. To make 6 cents with that set, a greedy algorithm would choose 3 coins {4, 1, 1} instead of the optimal solution that uses 2 coins {3, 3}. The general version of this problem is revisited later in Section 3.5.2 (Dynamic Programming).

UVa 410 - Station Balance (Load Balancing)

Given $1 \leq C \leq 5$ chambers which can store 0, 1, or 2 specimens, $1 \leq S \leq 2C$ specimens and a list M of the masses of the S specimens, determine which chamber should store each specimen in order to minimize 'imbalance'. See Figure 3.4 for a visual explanation[7].

---

[6]The presence of the 1-cent coin ensures that we can always make every value.

[7]Since $C \leq 5$ and $S \leq 10$, we can actually use a Complete Search solution for this problem. However, this problem is simpler to solve using the Greedy algorithm.

$A = (\sum_{j=1}^{S} M_j)/C$, i.e. A is the average of the total mass in each of the C chambers.

Imbalance $= \sum_{i=1}^{C} |X_i - A|$, i.e. the sum of differences between the total mass in each chamber w.r.t. A where X_i is the total mass of specimens in chamber i.

C = 3, S = 4, M = {5, 1, 2, 7}
Average mass / chamber
A = (5 + 1 + 2 + 7) / 3 = 5

IMBALANCE =
|(7+5)-5| + |2-5| + |1-5| =
7 + 3 + 4 = 14
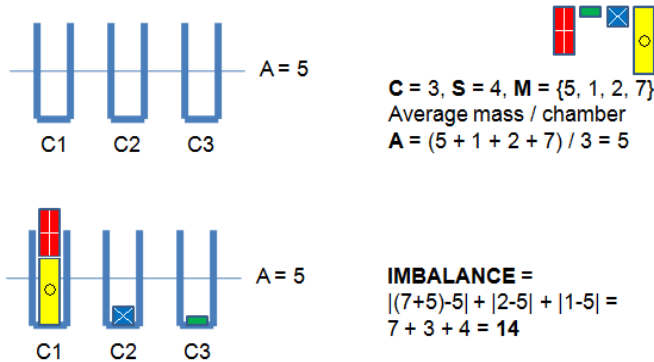
Figure 3.4: Visualization of UVa 410 - Station Balance

This problem can be solved using a greedy algorithm, but to arrive at that solution, we have to make several observations.

Imbalance too high in C3!

IMBALANCE =
|(7+1)-5| + |(2+5)-5| + |0-5| =
3 + 2 + 5 = 10

In this example, we already assign 3 specimens to 3 chambers, the 4th specimen must be paired with one of these 3 specimens already in their own chambers. Here, it is added to chamber 3.

IMBALANCE =
|7-5| + |2-5| + |(5+1)-5| =
2 + 3 + 1 = 6
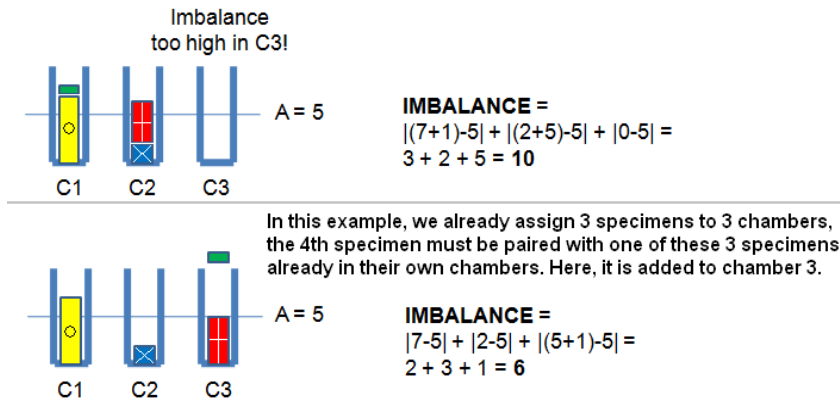
Figure 3.5: UVa 410 - Observations

Observation 1: If there exists an empty chamber, it is usually beneficial and never worse to move one specimen from a chamber with two specimens to the empty chamber! Otherwise, the empty chamber contributes more to the imbalance as shown in Figure 3.5, top.

Observation 2: If $S > C$, then $S - C$ specimens must be paired with a chamber already containing other specimens—the Pigeonhole principle! See Figure 3.5, bottom.

The key insight is that the solution to this problem can be simplified with sorting: if $S < 2C$, add $2C - S$ dummy specimens with mass 0. For example, $C = 3$, $S = 4$, $M = \{5, 1, 2, 7\} \rightarrow C = 3, S = 6, M = \{5, 1, 2, 7, 0, 0\}$. Then, sort the specimens on their mass such that $M_1 \leq M_2 \leq \ldots \leq M_{2C-1} \leq M_{2C}$. In this example, $M = \{5, 1, 2, 7, 0, 0\} \rightarrow \{0, 0, 1, 2, 5, 7\}$. By adding dummy specimens and then sorting them, a greedy strategy becomes 'apparent':

- Pair the specimens with masses $M_1 \& M_{2C}$ and put them in chamber 1, then

- Pair the specimens with masses $M_2 \& M_{2C-1}$ and put them in chamber 2, and so on ...

This greedy algorithm—known as *load balancing*—works! See Figure 3.6.

It is hard to impart the techniques used in deriving this greedy solution. Finding greedy solutions is an art, just as finding good Complete Search solutions requires creativity. A tip that arises from this example: If there is no obvious greedy strategy, try *sorting* the data or introducing some tweak and see if a greedy strategy emerges.
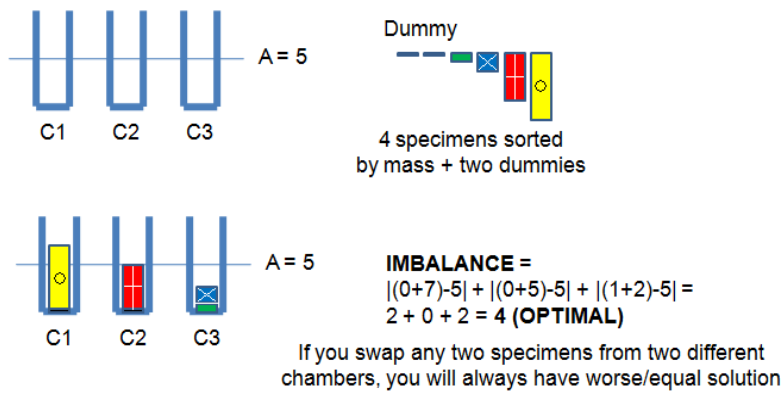
Figure 3.6: UVa 410 - Greedy Solution

UVa 10382 - Watering Grass (Interval Covering)

Problem description: n sprinklers are installed in a horizontal strip of grass L meters long and W meters wide. Each sprinkler is centered vertically in the strip. For each sprinkler, we are given its position as the distance from the left end of the center line and its radius of operation. What is the minimum number of sprinklers that should be turned on in order to water the entire strip of grass? Constraint: $n \le 10000$. For an illustration of the problem, see Figure 3.7—left side. The answer for this test case is 6 sprinklers (those labeled with {A, B, D, E, F, H}). There are 2 unused sprinklers: {C, G}.

We cannot solve this problem with a brute force strategy that tries all possible subsets of sprinklers to be turned on since the number of sprinklers can go up to 10000. It is definitely infeasible to try all 2^{10000} possible subsets of sprinklers.

This problem is actually a variant of the well known greedy problem called the *interval covering* problem. However, it includes a simple geometric twist. The original interval covering problem deals with intervals. This problem deals with sprinklers that have circles of influence in a horizontal area rather than simple intervals. We first have to transform the problem to resemble the standard interval covering problem.

See Figure 3.7—right side. We can convert these circles and horizontal strips into intervals. We can compute `dx = sqrt(`R^2` - (W/2)`^{2}`)`. Suppose a circle is centered at `(x, y)`. The interval represented by this circle is `[x-dx..x+dx]`. To see why this works, notice that the additional circle segment beyond `dx` away from `x` does not completely cover the strip in the horizontal region it spans. If you have difficulties with this geometric transformation, see Section 7.2.4 which discusses basic operations involving a *right triangle*.
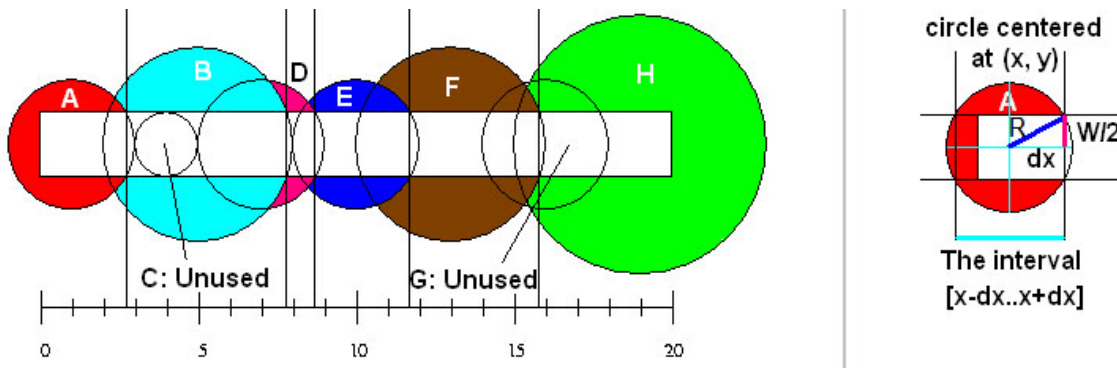
Figure 3.7: UVa 10382 - Watering Grass

Now that we have transformed the original problem into the interval covering problem, we can use the following Greedy algorithm. First, the Greedy algorithm sorts the intervals by *increasing* left endpoint and by *decreasing* right endpoint if ties arise. Then, the Greedy algorithm processes the intervals one at a time. It takes the interval that covers 'as far right as possible' and yet still produces uninterrupted coverage from the leftmost side to the rightmost side of the horizontal strip of grass. It ignores intervals that are already completely covered by other (previous) intervals.

For the test case shown in Figure 3.7—left side, this Greedy algorithm first sorts the intervals to obtain the sequence {A, B, C, D, E, F, G, H}. Then it processes them one by one. First, it takes 'A' (it has to), takes 'B' (connected to interval 'A'), ignores 'C' (as it is embedded inside interval 'B'), takes 'D' (it has to, as intervals 'B' and 'E' are not connected if 'D' is not used), takes 'E', takes 'F', ignores 'G' (as taking 'G' is not 'as far right as possible' and does not reach the rightmost side of the grass strip), takes 'H' (as it connects with interval 'F' and covers more to the right than interval of 'G' does, going beyond the rightmost end of the grass strip). In total, we select 6 sprinklers: {A, B, D, E, F, H}. This is the minimum possible number of sprinklers for this test case.

UVa 11292 - Dragon of Loowater (Sort the Input First)

Problem description: There are n dragon heads and m knights ($1 \leq n, m \leq 20000$). Each dragon head has a *diameter* and each knight has a *height*. A dragon head with diameter **D** can be chopped off by a knight with height **H** if **D** $\leq$ **H**. A knight can only chop off one dragon head. Given a list of diameters of the dragon heads and a list of heights of the knights, is it possible to chop off all the dragon heads? If yes, what is the minimum total height of the knights used to chop off the dragons' heads?

There are several ways to solve this problem, but we will illustrate one that is probably the easiest. This problem is a bipartite matching problem (this will be discussed in more detail in Section 4.7.4), in the sense that we are required to match (pair) certain knights to dragon heads in a maximal fashion. However, this problem can be solved greedily: Each dragon head should be chopped by a knight with the shortest height that is at least as tall as the diameter of the dragon's head. However, the input is given in an arbitrary order. If we sort both the list of dragon head diameters and knight heights in $O(n \log n + m \log m)$, we can use the following $O(min(n, m))$ scan to determine the answer. This is yet another example where sorting the input can help produce the required greedy strategy.

```
gold = d = k = 0; // array dragon+knight are sorted in non decreasing order
while (d < n && k < m) {                 // still have dragon heads or knights
  while (dragon[d] > knight[k] && k < m) k++;   // find the required knight
  if (k == m) break;      // no knight can kill this dragon head, doomed :S
  gold += knight[k];                       // the king pay this amount of gold
  d++; k++;                                 // next dragon head and knight please
}


if (d == n) printf("%d\n", gold);         // all dragon heads are chopped
else        printf("Loowater is doomed!\n");
```

Exercise 3.4.1.1*: Which of the following sets of coins (all in cents) are solvable using the greedy 'coin change' algorithm discussed in this section? If the greedy algorithm fails on a certain set of coin denominations, determine the smallest counter example V cents on which it fails to be optimal. See [51] for more details about finding such counter examples.

1. $S_1 = \{10, 7, 5, 4, 1\}$

2. $S_2 = \{64, 32, 16, 8, 4, 2, 1\}$

3. $S_3 = \{13, 11, 7, 5, 3, 2, 1\}$

4. $S_4 = \{7, 6, 5, 4, 3, 2, 1\}$

5. $S_5 = \{21, 17, 11, 10, 1\}$

Remarks About Greedy Algorithm in Programming Contests

In this section, we have discussed three classical problems solvable with Greedy algorithms: Coin Change (the special case), Load Balancing, and Interval Covering. For these classical problems, it is helpful to memorize their solutions (for this case, ignore that we have said earlier in the chapter about not relying too much on memorization). We have also discussed an important problem solving strategy usually applicable to greedy problems: Sorting the input data to elucidate hidden greedy strategies.

There are two other classical examples of Greedy algorithms in this book, e.g. Kruskal's (and Prim's) algorithm for the Minimum Spanning Tree (MST) problem (see Section 4.3) and Dijkstra's algorithm for the Single-Source Shortest Paths (SSSP) problem (see Section 4.4.3). There are many more known Greedy algorithms that we have chosen not to discuss in this book as they are too 'problem specific' and rarely appear in programming contests, e.g. Huffman Codes [7, 38], Fractional Knapsack [7, 38], some Job Scheduling problems, etc.

However, today's programming contests (both ICPC and IOI) rarely involve the purely canonical versions of these classical problems. Using Greedy algorithms to attack a 'non classical' problem is usually risky. A Greedy algorithm will normally not encounter the TLE response as it is often lightweight, but instead tends to obtain WA verdicts. Proving that a certain 'non-classical' problem has optimal sub-structure and greedy property during contest time may be difficult or time consuming, so a competitive programmer should usually use this rule of thumb:

If the input size is 'small enough' to accommodate the time complexity of either Complete Search or Dynamic Programming approaches (see Section 3.5), then use these approaches as both will ensure a correct answer. *Only* use a Greedy algorithm if the input size given in the problem statement are too large even for the best Complete Search or DP algorithm.

Having said that, it is increasingly true that problem authors try to set the input bounds of problems that allow for Greedy strategies to be in an ambiguous range so that contestants *cannot* use the input size to quickly determine the required algorithm!

We have to remark that it is quite challenging to come up with new 'non-classical' Greedy problems. Therefore, the number of such novel Greedy problems used in competitive programming is lower than that of Complete Search or Dynamic Programming problems.

Programming Exercises solvable using Greedy
(most hints are omitted to keep the problems challenging):

- Classical, Usually Easier

    1. UVa 00410 - Station Balance (discussed in this section, load balancing)

    2. UVa 01193 - Radar Installation (LA 2519, Beijing02, interval covering)

    3. UVa 10020 - Minimal Coverage (interval covering)

    4. UVa 10382 - Watering Grass (discussed in this section, interval covering)

    5. ***UVa 11264 - Coin Collector *** (coin change variant)

6. **UVa 11389 - The Bus Driver Problem *** (load balancing)
7. *UVa 12321 - Gas Station* (interval covering)
8. ***UVa 12405 - Scarecrow **** (simpler interval covering problem)
9. IOI 2011 - Elephants (optimized greedy solution can be used up to subtask 3, but the harder subtasks 4 and 5 must be solved using efficient data structure)

- Involving Sorting (Or The Input Is Already Sorted)

1. UVa 10026 - Shoemaker's Problem
2. *UVa 10037 - Bridge*
3. UVa 10249 - The Grand Dinner
4. UVa 10670 - Work Reduction
5. UVa 10763 - Foreign Exchange
6. UVa 10785 - The Mad Numerologist
7. ***UVa 11100 - The Trip, 2007 ****
8. UVa 11103 - WFF'N Proof
9. *UVa 11269 - Setting Problems*
10. **UVa 11292 - Dragon of Loowater ***
11. UVa 11369 - Shopaholic
12. UVa 11729 - Commando War
13. UVa 11900 - Boiled Eggs
14. ***UVa 12210 - A Match Making Problem ****
15. *UVa 12485 - Perfect Choir*

- Non Classical, Usually Harder

1. UVa 00311 - Packets
2. *UVa 00668 - Parliament*
3. UVa 10152 - ShellSort
4. UVa 10340 - All in All
5. UVa 10440 - Ferry Loading II
6. UVa 10602 - Editor Nottobad
7. **UVa 10656 - Maximum Sum (II) ***
8. UVa 10672 - Marbles on a tree
9. UVa 10700 - Camel Trading
10. UVa 10714 - Ants
11. **UVa 10718 - Bit Mask ***
12. *UVa 10982 - Troublemakers*
13. UVa 11054 - Wine Trading in Gergovia
14. **UVa 11157 - Dynamic Frog ***
15. *UVa 11230 - Annoying painting tool*
16. *UVa 11240 - Antimonotonicity*
17. *UVa 11335 - Discrete Pursuit*
18. UVa 11520 - Fill the Square
19. UVa 11532 - Simple Adjacency ...
20. UVa 11567 - Moliu Number Generator
21. *UVa 12482 - Short Story Competition*

3.5 Dynamic Programming

Dynamic Programming (from now on abbreviated as DP) is perhaps the most challenging problem-solving technique among the four paradigms discussed in this chapter. Thus, make sure that you have mastered the material mentioned in the previous chapters/sections before reading this section. Also, prepare to see lots of recursion and recurrence relations!

The key skills that you have to develop in order to master DP are the abilities to determine the problem *state*s and to determine the relationships or *transitions* between current problems and their sub-problems. We have used these skills earlier in recursive backtracking (see Section 3.2.2). In fact, DP problems with small input size constraints may already be solvable with recursive backtracking.

If you are new to DP technique, you can start by assuming that (the 'top-down') DP is a kind of 'intelligent' or 'faster' recursive backtracking. In this section, we will explain the reasons why DP is often faster than recursive backtracking for problems amenable to it.

DP is primarily used to solve *optimization* problems and *counting* problems. If you encounter a problem that says "minimize this" or "maximize that" or "count the ways to do that", then there is a (high) chance that it is a DP problem. Most DP problems in programming contests only ask for the optimal/total value and not the optimal solution itself, which often makes the problem easier to solve by removing the need to backtrack and produce the solution. However, some harder DP problems also require the optimal solution to be returned in some fashion. We will continually refine our understanding of Dynamic Programming in this section.

3.5.1 DP Illustration

We will illustrate the concept of Dynamic Programming with an example problem: UVa 11450 - Wedding Shopping. The abridged problem statement: Given different options for each garment (e.g. 3 shirt models, 2 belt models, 4 shoe models, ...) and a certain *limited* budget, our task is to *buy one model of each garment*. We cannot spend more money than the given budget, but we want to spend *the maximum possible* amount.

The input consists of two integers $1 \leq M \leq 200$ and $1 \leq C \leq 20$, where M is the budget and C is the number of garments that you have to buy, followed by some information about the C garments. For the garment $g \in [0..C-1]$, we will receive an integer $1 \leq K \leq 20$ which indicates the number of different models there are for that garment g, followed by K integers indicating the price of each model $\in [1..K]$ of that garment g.

The output is one integer that indicates the maximum amount of money we can spend purchasing one of each garment *without exceeding the budget*. If there is no solution due to the small budget given to us, then simply print "no solution".

Suppose we have the following test case A with $M = 20$, $C = 3$:
Price of the 3 models of garment g = 0 → 6 4 <u>8</u> // the prices are not sorted in the input
Price of the 2 models of garment g = 1 → 5 <u>10</u>
Price of the 4 models of garment g = 2 → <u>1</u> 5 3 5

For this test case, the answer is 19, which *may* result from buying the <u>underlined</u> items (8+10+1). This is not unique, as solutions (6+10+3) and (4+10+5) are also optimal.

However, suppose we have this test case B with $M = \mathbf{9}$ **(limited budget)**, $C = 3$:
Price of the 3 models of garment g = 0 → 6 4 8
Price of the 2 models of garment g = 1 → 5 10
Price of the 4 models of garment g = 2 → 1 5 3 5

The answer is then "`no solution`" because even if we buy all the cheapest models for each garment, the total price $(4+5+1) = 10$ still exceeds our given budget $M = 9$.

In order for us to appreciate the usefulness of Dynamic Programming in solving the above-mentioned problem, let's explore how far the *other* approaches discussed earlier will get us in this particular problem.

Approach 1: Greedy (Wrong Answer)

Since we want to maximize the budget spent, one greedy idea (there are other greedy approaches—which are also WA) is to take the most expensive model for each garment g which still fits our budget. For example in test case A above, we can choose the most expensive model 3 of garment `g = 0` with price 8 (`money` is now `20-8 = 12`), then choose the most expensive model 2 of garment `g = 1` with price 10 (`money = 12-10 = 2`), and finally for the last garment `g = 2`, we can only choose model 1 with price 1 as the `money` we have left does not allow us to buy the other models with price 3 or 5. This greedy strategy 'works' for test cases A and B above and produce the same optimal solution $(8+10+1) = 19$ and "`no solution`", respectively. It also runs very fast[8]: $20 + 20 + \ldots + 20$ for a total of 20 times $= 400$ operations in the worst case. However, this greedy strategy does not work for many other test cases, such as this *counter-example* below (test case C):

Test case C with $M = 12$, $C = 3$:
3 models of garment `g = 0` → 6 <u>4</u> **8**
2 models of garment `g = 1` → <u>5</u> 10
4 models of garment `g = 2` → 1 5 <u>3</u> 5

The Greedy strategy selects model 3 of garment `g = 0` with price **8** (`money = 12-8 = 4`), causing us to not have enough money to buy any model in garment `g = 1`, thus incorrectly reporting "`no solution`". One optimal solution is <u>4+5+3</u> $= 12$, which uses up all of our budget. The optimal solution is not unique as $6+5+1 = 12$ also depletes the budget.

Approach 2: Divide and Conquer (Wrong Answer)

This problem is not solvable using the Divide and Conquer paradigm. This is because the sub-problems (explained in the Complete Search sub-section below) are not independent. Therefore, we cannot solve them separately with the Divide and Conquer approach.

Approach 3: Complete Search (Time Limit Exceeded)

Next, let's see if Complete Search (recursive backtracking) can solve this problem. One way to use recursive backtracking in this problem is to write a function `shop(money, g)` with two parameters: The current `money` that we have and the current garment `g` that we are dealing with. The pair (`money, g`) is the *state* of this problem. Note that the order of parameters does not matter, e.g. (`g, money`) is also a perfectly valid state. Later in Section 3.5.3, we will see more discussion on how to select appropriate states for a problem.

We start with `money = M` and garment `g = 0`. Then, we try all possible models in garment `g = 0` (a maximum of 20 models). If model i is chosen, we subtract model i's price from `money`, then repeat the process in a recursive fashion with garment `g = 1` (which can also have up to 20 models), etc. We stop when the model for the last garment `g = C-1` has been chosen. If `money < 0` before we choose a model from garment `g = C-1`, we can prune the infeasible solution. Among all valid combinations, we can then pick the one that results in the smallest non-negative `money`. This maximizes the money spent, which is (`M - money`).

---

[8]We do not need to sort the prices just to find the model with the maximum price as there are only up to $K \leq 20$ models. An $O(K)$ scan is enough.

We can formally define these Complete Search recurrences (transitions) as follows:

1. If `money < 0` (i.e. money goes negative),
`shop(money, g)` = $-\infty$ (in practice, we can just return a large negative value)

2. If a model from the last garment has been bought, that is, `g = C`,
`shop(money, g) = M - money` (this is the actual money that we spent)

3. In general case, $\forall$ `model` $\in$ `[1..K]` of current garment `g`,
`shop(money, g) = max(shop(money - price[g][model], g + 1))`
We want to maximize this value (Recall that the invalid ones have large negative value)

This solution works correctly, but it is **very slow**! Let's analyze the worst case time complexity. In the largest test case, garment `g = 0` has up to 20 models; garment `g = 1` *also* has up to 20 models and all garments including the last garment `g = 19` ***also*** have up to 20 models. Therefore, this Complete Search runs in $20 \times 20 \times \ldots \times 20$ operations in the worst case, i.e. 20^{20} = a **very large** number. If we can *only* come up with this Complete Search solution, we cannot solve this problem.

Approach 4: Top-Down DP (Accepted)

To solve this problem, we have to use the DP concept as this problem satisfies the two prerequisites for DP to be applicable:

1. This problem has optimal sub-structures[9].
   This is illustrated in the third Complete Search recurrence above: The solution for the sub-problem is part of the solution of the original problem. In other words, if we select model i for garment $g = 0$, for our final selection to be optimal, our choice for garments $g = 1$ and above must also be the optimal choice for a reduced budget of $M - price$, where *price* refers to the price of model i.

2. This problem has overlapping sub-problems.
   This is the key characteristic of DP! The search space of this problem is *not* as big as the rough 20^{20} bound obtained earlier because **many** sub-problems are *overlapping*!

Let's verify if this problem indeed has overlapping sub-problems. Suppose that there are 2 models in a certain garment `g` with the *same* price `p`. Then, a Complete Search will move to the **same** sub-problem `shop(money - p, g + 1)` after picking *either* model! This situation will also occur if some combination of `money` and chosen model's price causes `money`$_1$ `- p`$_1$ `= money`$_2$ `- p`$_2$ at the same garment `g`. This will—in a Complete Search solution—cause the same sub-problem to be computed *more than once*, an inefficient state of affairs!

   So, how many *distinct* sub-problems (a.k.a. **states** in DP terminology) are there in this problem? Only $201 \times 20 = 4020$. There are only 201 possible values for `money` (0 to 200 inclusive) and 20 possible values for the garment `g` (0 to 19 inclusive). Each sub-problem just needs to be computed *once*. If we can ensure this, we can solve this problem *much faster*.

   The implementation of this DP solution is surprisingly simple. If we already have the recursive backtracking solution (see the recurrences—a.k.a. **transitions** in DP terminology—shown in the Complete Search approach above), we can implement the **top-down** DP by adding these two additional steps:

1. Initialize[10] a DP 'memo' table with dummy values that are not used in the problem, e.g. '-1'. The DP table should have dimensions corresponding to the problem states.

---

[9]Optimal sub-structures are also required for Greedy algorithms to work, but this problem lacks the 'greedy property', making it unsolvable with the Greedy algorithm.

[10]For C/C++ users, the `memset` function in `<cstring>` is a good tool to perform this step.

2. At the start of the recursive function, check if this state has been computed before.

   (a) If it has, simply return the value from the DP memo table, $O(1)$.
       (This the origin of the term 'memoization'.)

   (b) If it has not been computed, perform the computation as per normal (only once) and then store the computed value in the DP memo table so that *further calls* to this sub-problem (state) return immediately.

Analyzing a basic[11] DP solution is easy. If it has M distinct states, then it requires $O(M)$ memory space. If computing one state (the complexity of the DP transition) requires $O(k)$ steps, then the overall time complexity is $O(kM)$. This UVa 11450 problem has $M = 201 \times 20 = 4020$ and $k = 20$ (as we have to iterate through at most 20 models for each garment g). Thus, the time complexity is at most $4020 \times 20 = 80400$ operations per test case, a very manageable calculation.

We display our code below for illustration, especially for those who have never coded a top-down DP algorithm before. Scrutinize this code and verify that it is indeed very similar to the recursive backtracking code that you have seen in Section 3.2.

```
/* UVa 11450 - Wedding Shopping - Top Down */
// assume that the necessary library files have been included
// this code is similar to recursive backtracking code
// parts of the code specific to top-down DP are commented with: 'TOP-DOWN'

int M, C, price[25][25];                  // price[g (<= 20)][model (<= 20)]
int memo[210][25];    // TOP-DOWN: dp table memo[money (<= 200)][g (<= 20)]
int shop(int money, int g) {
  if (money < 0) return -1000000000;     // fail, return a large -ve number
  if (g == C) return M - money;          // we have bought last garment, done
  // if the line below is commented, top-down DP will become backtracking!!
  if (memo[money][g] != -1) return memo[money][g]; // TOP-DOWN: memoization
  int ans = -1;   // start with a -ve number as all prices are non negative
  for (int model = 1; model <= price[g][0]; model++)      // try all models
    ans = max(ans, shop(money - price[g][model], g + 1));
  return memo[money][g] = ans; }     // TOP-DOWN: memoize ans and return it

int main() {             // easy to code if you are already familiar with it
  int i, j, TC, score;
  scanf("%d", &TC);
  while (TC--) {
    scanf("%d %d", &M, &C);
    for (i = 0; i < C; i++) {
      scanf("%d", &price[i][0]);                    // store K in price[i][0]
      for (j = 1; j <= price[i][0]; j++) scanf("%d", &price[i][j]);
    }
    memset(memo, -1, sizeof memo);    // TOP-DOWN: initialize DP memo table
    score = shop(M, 0);                           // start the top-down DP
    if (score < 0) printf("no solution\n");
    else          printf("%d\n", score);
} } // return 0;
```

[11]Basic means "without fancy optimizations that we will see later in this section and in Section 8.3".

We want to take this opportunity to illustrate another style used in implementing DP solutions (only applicable for C/C++ users). Instead of frequently addressing a certain cell in the memo table, we can use a local *reference* variable to store the memory address of the required cell in the memo table as shown below. The two coding styles are not very different, and it is up to you to decide which style you prefer.

```
int shop(int money, int g) {
  if (money < 0) return -1000000000; // order of >1 base cases is important
  if (g == C) return M - money;  // money can't be <0 if we reach this line
  int &ans = memo[money][g];                    // remember the memory address
  if (ans != -1) return ans;
  for (int model = 1; model <= price[g][0]; model++)
    ans = max(ans, shop(money - price[g][model], g + 1));
  return ans;                  // ans (or memo[money][g]) is directly updated
}
```

Source code: `ch3_02_UVa11450_td.cpp/java`

Approach 5: Bottom-Up DP (Accepted)

There is another way to implement a DP solution often referred to as the **bottom-up** DP. This is actually the 'true form' of DP as DP was originally known as the 'tabular method' (computation technique involving a table). The *basic* steps to build bottom-up DP solution are as follows:

1. Determine the required set of parameters that uniquely describe the problem (the state). This step is similar to what we have discussed in recursive backtracking and top-down DP earlier.

2. If there are N parameters required to represent the states, prepare an N dimensional DP table, with one entry per state. This is equivalent to the memo table in top-down DP. However, there are differences. In bottom-up DP, we only need to initialize some cells of the DP table with known initial values (the base cases). Recall that in top-down DP, we initialize the memo table completely with dummy values (usually -1) to indicate that we have not yet computed the values.

3. Now, with the base-case cells/states in the DP table already filled, determine the cells/states that can be filled next (the transitions). Repeat this process until the DP table is complete. For the bottom-up DP, this part is usually accomplished through iterations, using loops (more details about this later).

For UVa 11450, we can write the bottom-up DP as follow: We describe the state of a sub-problem with two parameters: The current garment `g` and the current `money`. This state formulation is essentially equivalent to the state in the top-down DP above, except that we have reversed the order to make `g` the first parameter (thus the values of `g` are the row indices of the DP table so that we can take advantage of cache-friendly row-major traversal in a 2D array, see the speed-up tips in Section 3.2.3). Then, we initialize a 2D table (boolean matrix) `reachable[g][money]` of size 20×201. Initially, only cells/states reachable by buying any of the models of the first garment `g = 0` are set to true (in the first row). Let's use test case A above as example. In Figure 3.8, top, the only columns '20-6 = 14', '20-4 = 16', and '20-8 = 12' in row 0 are initially set to true.

money =>

g ‖ ∨	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	1	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	1	0	0	0
1	0	0	1	0	1	0	1	1	0	1	0	1	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	1	0	0	0
1	0	0	1	0	1	0	1	1	0	1	0	1	0	0	0	0	0	0	0	0	0
2	0	1	1	1	1	1	1	0	1	0	1	0	0	0	0	0	0	0	0	0	0

Figure 3.8: Bottom-Up DP (columns 21 to 200 are not shown)

Now, we loop from the second garment `g = 1` (second row) to the last garment `g = C-1 = 3-1 = 2` (third and last row) in row-major order (row by row). If `reachable[g-1][money]` is true, then the next state `reachable[g][money-p]` where `p` is the price of a model of current garment `g` is also reachable as long as the second parameter (`money`) is not negative. See Figure 3.8, middle, where `reachable[0][16]` propagates to `reachable[1][16-5]` and `reachable[1][16-10]` when the model with price 5 and 10 in garment `g = 1` is bought, respectively; `reachable[0][12]` propagates to `reachable[1][12-10]` when the model with price 10 in garment `g = 1` is bought, etc. We repeat this table filling process row by row until we are done with the last row[12].

Finally, the answer can be found in the last row when `g = C-1`. Find the state in that row that is both nearest to index 0 and reachable. In Figure 3.8, bottom, the cell `reachable[2][1]` provides the answer. This means that we can reach state (`money = 1`) by buying some combination of the various garment models. The required final answer is actually `M - money`, or in this case, `20-1 = 19`. The answer is "`no solution`" if there is no state in the last row that is reachable (where `reachable[C-1][money]` is set to true). We provide our implementation below for comparison with the top-down version.

```
/* UVa 11450 - Wedding Shopping - Bottom Up */
// assume that the necessary library files have been included

int main() {
  int g, money, k, TC, M, C;
  int price[25][25];                      // price[g (<= 20)][model (<= 20)]
  bool reachable[25][210];    // reachable table[g (<= 20)][money (<= 200)]

  scanf("%d", &TC);
  while (TC--) {
    scanf("%d %d", &M, &C);
    for (g = 0; g < C; g++) {
      scanf("%d", &price[g][0]);               // we store K in price[g][0]
      for (money = 1; money <= price[g][0]; money++)
        scanf("%d", &price[g][money]);
    }
```

---

[12]Later in Section 4.7.1, we will discuss DP as a traversal of an (implicit) DAG. To avoid unnecessary 'backtracking' along this DAG, we have to visit the vertices in their topological order (see Section 4.2.5). The order in which we fill the DP table is a topological ordering of the underlying implicit DAG.

```
    memset(reachable, false, sizeof reachable);         // clear everything
    for (g = 1; g <= price[0][0]; g++)      // initial values (base cases)
      if (M - price[0][g] >= 0)        // to prevent array index out of bound
        reachable[0][M - price[0][g]] = true;  // using first garment g = 0

    for (g = 1; g < C; g++)                          // for each remaining garment
      for (money = 0; money < M; money++) if (reachable[g-1][money])
        for (k = 1; k <= price[g][0]; k++) if (money - price[g][k] >= 0)
          reachable[g][money - price[g][k]] = true;   // also reachable now

    for (money = 0; money <= M && !reachable[C - 1][money]; money++);

    if (money == M + 1) printf("no solution\n");  // last row has no on bit
    else                printf("%d\n", M - money);
  }
} // return 0;
```

Source code: ch3_03_UVa11450_bu.cpp/java

There is an advantage for writing DP solutions in the bottom-up fashion. For problems where we only need the last row of the DP table (or, more generally, the last updated slice of all the states) to determine the solution—including this problem, we can optimize the *memory usage* of our DP solution by sacrificing one dimension in our DP table. For harder DP problems with tight memory requirements, this 'space saving trick' may prove to be useful, though the overall time complexity does not change.

Let's take a look again at Figure 3.8. We only need to store two rows, the current row we are processing and the previous row we have processed. To compute row 1, we only need to know the columns in row 0 that are set to true in `reachable`. To compute row 2, we similarly only need to know the columns in row 1 that are set to true in `reachable`. In general, to compute row g, we only need values from the previous row $g - 1$. So, instead of storing a boolean matrix `reachable[g][money]` of size $\underline{20} \times 201$, we can simply store `reachable[2][money]` of size $\underline{2} \times 201$. We can use this programming trick to reference one row as the 'previous' row and another row as the 'current' row (e.g. `prev = 0, cur = 1`) and then swap them (e.g. now `prev = 1, cur = 0`) as we compute the bottom-up DP row by row. Note that for this problem, the memory savings are not significant. For harder DP problems, for example where there might be thousands of garment models instead of 20, this space saving trick can be important.

Top-Down versus Bottom-Up DP

Although both styles use 'tables', the way the bottom-up DP table is filled is different to that of the top-down DP *memo* table. In the top-down DP, the memo table entries are filled 'as needed' through the recursion itself. In the bottom-up DP, we used a correct 'DP table filling order' to compute the values such that the previous values needed to process the current cell have already been obtained. This table filling order is the topological order of the implicit DAG (this will be explained in more detail in Section 4.7.1) in the recurrence structure. For most DP problems, a topological order can be achieved simply with the proper sequencing of some (nested) loops.

For most DP problems, these two styles are equally good and the decision to use a particular DP style is a matter of preference. However, for harder DP problems, one of the

styles can be better than the other. To help you understand which style that you should use when presented with a DP problem, please study the trade-offs between top-down and bottom-up DPs listed in Table 3.2.

Top-Down	Bottom-Up
Pros: 1. It is a natural transformation from the normal Complete Search recursion 2. Computes the sub-problems only when necessary (sometimes this is faster)	Pros: 1. Faster if many sub-problems are revisited as there is no overhead from recursive calls 2. Can save memory space with the 'space saving trick' technique
Cons: 1. Slower if many sub-problems are revisited due to function call overhead (this is not usually penalized in programming contests) 2. If there are M states, an $O(M)$ table size is required, which can lead to MLE for some harder problems (except if we use the trick in Section 8.3.4)	Cons: 1. For programmers who are inclined to recursion, this style may not be intuitive 2. If there are M states, bottom-up DP visits and fills the value of *all* these M states

Table 3.2: DP Decision Table

Displaying the Optimal Solution

Many DP problems request only for the value of the optimal solution (like the UVa 11450 above). However, many contestants are caught off-guard when they are also required to print the optimal solution. We are aware of two ways to do this.

The first way is mainly used in the bottom-up DP approach (which is still applicable for top-down DPs) where we store the predecessor information at each state. If there are more than one optimal predecessors and we have to output all optimal solutions, we can store those predecessors in a list. Once we have the optimal final state, we can do backtracking from the optimal final state and follow the optimal transition(s) recorded at each state until we reach one of the base cases. If the problem asks for all optimal solutions, this backtracking routine will print them all. However, most problem authors usually set additional output criteria so that the selected optimal solution is unique (for easier judging).

Example: See Figure 3.8, bottom. The optimal final state is `reachable[2][1]`. The predecessor of this optimal final state is state `reachable[1][2]`. We now backtrack to `reachable[1][2]`. Next, see Figure 3.8, middle. The predecessor of state `reachable[1][2]` is state `reachable[0][12]`. We then backtrack to `reachable[0][12]`. As this is already one of the initial base states (at the first row), we know that an optimal solution is: (20→12) = price 8, then (12→2) = price 10, then (2→1) = price 1. However, as mentioned earlier in the problem description, this problem may have several other optimal solutions, e.g. We can also follow the path: `reachable[2][1]` → `reachable[1][6]` → `reachable[0][16]` which represents another optimal solution: (20→16) = price 4, then (16→6) = price 10, then (6→1) = price 5.

The second way is applicable mainly to the top-down DP approach where we utilize the strength of recursion and memoization to do the same job. Using the top-down DP code shown in Approach 4 above, we will add another function `void print_shop(int money, int g)` that has the same structure as `int shop(int money, int g)` except that it uses the values stored in the memo table to reconstruct the solution. A sample implementation (that only prints out one optimal solution) is shown below:

```
void print_shop(int money, int g) {          // this function returns void
  if (money < 0 || g == C) return;                // similar base cases
  for (int model = 1; model <= price[g][0]; model++) // which model is opt?
    if (shop(money - price[g][model], g + 1) == memo[money][g]) {
      printf("%d%c", price[g][model], g == C-1 ? '\n' : '-');   // this one
      print_shop(money - price[g][model], g + 1);  // recurse to this state
      break;                                   // do not visit other states
}    }
```

Exercise 3.5.1.1: To verify your understanding of UVa 11450 problem discussed in this section, determine what is the output for test case D below?

Test case D with $M = 25$, $C = 3$:
Price of the 3 models of garment g = 0 $\rightarrow$ 6 4 8
Price of the 2 models of garment g = 1 $\rightarrow$ 10 6
Price of the 4 models of garment g = 2 $\rightarrow$ 7 3 1 5

Exercise 3.5.1.2: Is the following state formulation shop(g, model), where g represents the current garment and model represents the current model, appropriate and exhaustive for UVa 11450 problem?

Exercise 3.5.1.3: Add the space saving trick to the bottom-up DP code in Approach 5!

3.5.2 Classical Examples

The problem UVa 11450 - Wedding Shopping above is a (relatively easy) non-classical DP problem, where we had to come up with the correct DP states and transitions *by ourself*. However, there are many other *classical* problems with efficient DP solutions, i.e. their DP states and transitions are *well-known*. Therefore, such classical DP problems and their solutions should be mastered by every contestant who wishes to do well in ICPC or IOI! In this section, we list down six classical DP problems and their solutions. Note: Once you understand the basic form of these DP solutions, try solving the programming exercises that enumerate their *variants*.

1. Max 1D Range Sum

Abridged problem statement of UVa 507 - Jill Rides Again: Given an integer array A containing $n \leq 20K$ non-zero integers, determine the maximum (1D) range sum of A. In other words, find the maximum Range Sum Query (RSQ) between two indices i and j in [0..n-1], that is: A[i] + A[i+1] + A[i+2] +...+ A[j] (also see Section 2.4.3 and 2.4.4).

A Complete Search algorithm that tries all possible $O(n^2)$ pairs of i and j, computes the required RSQ(i, j) in $O(n)$, and finally picks the maximum one runs in an overall time complexity of $O(n^3)$. With n up to $20K$, this is a TLE solution.

In Section 2.4.4, we have discussed the following DP strategy: Pre-process array A by computing A[i] += A[i-1] $\forall i \in$ [1..n-1] so that A[i] contains the sum of integers in subarray A[0..i]. We can now compute RSQ(i, j) in $O(1)$: RSQ(0, j) = A[j] and RSQ(i, j) = A[j] - A[i-1] $\forall i > 0$. With this, the Complete Search algorithm above can be made to run in $O(n^2)$. For n up to $20K$, this is still a TLE approach. However, this technique is still useful in other cases (see the usage of this 1D Range Sum in Section 8.4.2).

There is an even better algorithm for this problem. The main part of Jay Kadane's $O(n)$ (can be viewed as a greedy or DP) algorithm to solve this problem is shown below.

```
// inside int main()
  int n = 9, A[] = { 4, -5, 4, -3, 4, 4, -4, 4, -5 };    // a sample array A
  int sum = 0, ans = 0;            // important, ans must be initialized to 0
  for (int i = 0; i < n; i++) {                          // linear scan, O(n)
    sum += A[i];                       // we greedily extend this running sum
    ans = max(ans, sum);                  // we keep the maximum RSQ overall
    if (sum < 0) sum = 0;                    // but we reset the running sum
  }                                          // if it ever dips below 0
  printf("Max 1D Range Sum = %d\n", ans);
```

Source code: ch3_04_Max1DRangeSum.cpp/java

The key idea of Kadane's algorithm is to keep a running sum of the integers seen so far and greedily reset that to 0 if the running sum dips below 0. This is because re-starting from 0 is always better than continuing from a negative running sum. Kadane's algorithm is the required algorithm to solve this UVa 507 problem as $n \leq 20K$.

Note that we can also view this Kadane's algorithm as a DP solution. At each step, we have two choices: We can either leverage the previously accumulated maximum sum, or begin a new range. The DP variable `dp(i)` thus represents the maximum sum of a range of integers that ends with element `A[i]`. Thus, the final answer is the maximum over all the values of `dp(i)` where `i ∈ [0..n-1]`. If zero-length ranges are allowed, then 0 must also be considered as a possible answer. The implementation above is essentially an efficient version that utilizes the space saving trick discussed earlier.

2. Max 2D Range Sum

Abridged problem statement of UVa 108 - Maximum Sum: Given an $n \times n$ $(1 \leq n \leq 100)$ square matrix of integers A where each integer ranges from [-127..127], find a sub-matrix of A with the maximum sum. For example: The 4×4 matrix $(n = 4)$ in Table 3.3.A below has a 3×2 sub-matrix on the lower-left with maximum sum of 9 + 2 - 4 + 1 - 1 + 8 = 15.

Table 3.3: UVa 108 - Maximum Sum

Attacking this problem naïvely using a Complete Search as shown below does not work as it runs in $O(n^6)$. For the largest test case with $n = 100$, an $O(n^6)$ algorithm is too slow.

```
maxSubRect = -127*100*100;     // the lowest possible value for this problem
for (int i = 0; i < n; i++) for (int j = 0; j < n; j++) // start coordinate
  for (int k = i; k < n; k++) for (int l = j; l < n; l++) {    // end coord
    subRect = 0;                         // sum the items in this sub-rectangle
    for (int a = i; a <= k; a++) for (int b = j; b <= l; b++)
      subRect += A[a][b];
    maxSubRect = max(maxSubRect, subRect); }           // the answer is here
```

The solution for the Max 1D Range Sum in the previous subsection can be extended to two (or more) dimensions as long as the inclusion-exclusion principle is properly applied. The only difference is that while we dealt with overlapping sub-ranges in Max 1D Range Sum, we will deal with overlapping sub-matrices in Max 2D Range Sum. We can turn the $n \times n$ input matrix into an $n \times n$ *cumulative sum matrix* where `A[i][j]` no longer contains its own value, but the sum of all items within sub-matrix `(0, 0)` to `(i, j)`. This can be done simultaneously while reading the input and still runs in $O(n^2)$. The code shown below turns the input square matrix (see Table 3.3.A) into a cumulative sum matrix (see Table 3.3.B).

```
scanf("%d", &n);                         // the dimension of input square matrix
for (int i = 0; i < n; i++) for (int j = 0; j < n; j++) {
  scanf("%d", &A[i][j]);
  if (i > 0) A[i][j] += A[i - 1][j];          // if possible, add from top
  if (j > 0) A[i][j] += A[i][j - 1];          // if possible, add from left
  if (i > 0 && j > 0) A[i][j] -= A[i - 1][j - 1];    // avoid double count
}                                          // inclusion-exclusion principle
```

With the sum matrix, we can answer the sum of any sub-matrix `(i, j)` to `(k, l)` in $O(1)$ using the code below. For example, let's compute the sum of `(1, 2)` to `(3, 3)`. We split the sum into 4 parts and compute `A[3][3] - A[0][3] - A[3][1] + A[0][1] = -3 - 13 - (-9) + (-2) = -9` as highlighted in Table 3.3.C. With this $O(1)$ DP formulation, the Max 2D Range Sum problem can now be solved in $O(n^4)$. For the largest test case of UVa 108 with $n = 100$, this is still fast enough.

```
maxSubRect = -127*100*100;     // the lowest possible value for this problem
for (int i = 0; i < n; i++) for (int j = 0; j < n; j++) // start coordinate
  for (int k = i; k < n; k++) for (int l = j; l < n; l++) {     // end coord
    subRect = A[k][l];        // sum of all items from (0, 0) to (k, l): O(1)
    if (i > 0) subRect -= A[i - 1][l];                             // O(1)
    if (j > 0) subRect -= A[k][j - 1];                             // O(1)
    if (i > 0 && j > 0) subRect += A[i - 1][j - 1];                // O(1)
    maxSubRect = max(maxSubRect, subRect); }        // the answer is here
```

Source code: `ch3_05_UVa108.cpp/java`

From these two examples—the Max 1D and 2D Range Sum Problems—we can see that not every range problem requires a Segment Tree or a Fenwick Tree as discussed in Section 2.4.3 or 2.4.4. Static-input range-related problems are often solvable with DP techniques. It is also worth mentioning that the solution for a range problem is very natural to produce with bottom-up DP techniques as the operand is already a 1D or a 2D array. We can still write the recursive top-down solution for a range problem, but the solution is not as natural.

3. Longest Increasing Subsequence (LIS)

Given a sequence $\{$`A[0]`, `A[1]`,..., `A[n-1]`$\}$, determine its Longest Increasing Subsequence (LIS)[13]. Note that these 'subsequences' are not necessarily contiguous. Example: $n = 8$, $A = \{\underline{-7}, 10, 9, \underline{2, 3, 8}, 8, 1\}$. The length-4 LIS is {-7, 2, 3, 8}.

---

[13]There are other variants of this problem, including the Longest *Decreasing* Subsequence and Longest *Non Increasing/Decreasing* Subsequence. The increasing subsequences can be modeled as a Directed Acyclic Graph (DAG) and finding the LIS is equivalent to finding the Longest Paths in the DAG (see Section 4.7.1).

Index	0	1	2	3	4	5	6	7
A	-7	10	9	2	3	8	8	1
LIS(i)	1	2	2	2	3	4	4	2

Figure 3.9: Longest Increasing Subsequence

As mentioned in Section 3.1, a naïve Complete Search that enumerates all possible subsequences to find the longest increasing one is too slow as there are $O(2^n)$ possible subsequences. Instead of trying all possible subsequences, we can consider the problem with a different approach. We can write the state of this problem with just one parameter: i. Let LIS(i) be the LIS ending at index i. We know that LIS(0) = 1 as the first number in A is itself a subsequence. For i ≥ 1, LIS(i) is slightly more complex. We need to find the index j such that j < i and A[j] < A[i] and LIS(j) is the largest. Once we have found this index j, we know that LIS(i) = 1 + LIS(j). We can write this recurrence formally as:

1. LIS(0) = 1 // the base case
2. LIS(i) = max(LIS(j) + 1), ∀j ∈ [0..i-1] and A[j] < A[i] // the recursive case, one more than the previous best solution ending at j for all j < i.

The answer is the largest value of LIS(k) ∀k ∈ [0..n-1].

Now let's see how this algorithm works (also see Figure 3.9):

- LIS(0) is 1, the first number in A = {-7}, the base case.

- LIS(1) is 2, as we can extend LIS(0) = {-7} with {10} to form {-7, 10} of length 2. The best j for i = 1 is j = 0.

- LIS(2) is 2, as we can extend LIS(0) = {-7} with {9} to form {-7, 9} of length 2. We cannot extend LIS(1) = {-7, 10} with {9} as it is non increasing. The best j for i = 2 is j = 0.

- LIS(3) is 2, as we can extend LIS(0) = {-7} with {2} to form {-7, 2} of length 2. We cannot extend LIS(1) = {-7, 10} with {2} as it is non-increasing. We also cannot extend LIS(2) = {-7, 9} with {2} as it is also non-increasing. The best j for i = 3 is j = 0.

- LIS(4) is 3, as we can extend LIS(3) = {-7, 2} with {3} to form {-7, 2, 3}. This is the best choice among the possibilities. The best j for i = 4 is j = 3.

- LIS(5) is 4, as we can extend LIS(4) = {-7, 2, 3} with {8} to form {-7, 2, 3, 8}. This is the best choice among the possibilities. The best j for i = 5 is j = 4.

- LIS(6) is 4, as we can extend LIS(4) = {-7, 2, 3} with {8} to form {-7, 2, 3, 8}. This is the best choice among the possibilities. The best j for i = 6 is j = 4.

- LIS(7) is 2, as we can extend LIS(0) = {-7} with {1} to form {-7, 1}. This is the best choice among the possibilities. The best j for i = 7 is j = 0.

- The answers lie at LIS(5) or LIS(6); both values (LIS lengths) are 4. See that the index k where LIS(k) is the highest can be anywhere in [0..n-1].

106

There are clearly many overlapping sub-problems in LIS problem because to compute `LIS(i)`, we need to compute `LIS(j)` $\forall$`j` $\in$ `[0..i-1]`. However, there are only n distinct states, the indices of the LIS ending at index `i`, $\forall$`i` $\in$ `[0..n-1]`. As we need to compute each state with an $O(n)$ loop, this DP algorithm runs in $O(n^2)$.

If needed, the LIS solution(s) can be reconstructed by storing the predecessor information (the arrows in Figure 3.9) and tracing the arrows from index `k` that contain the highest value of `LIS(k)`. For example, `LIS(5)` is the optimal final state. Check Figure 3.9. We can trace the arrows as follow: `LIS(5)` $\rightarrow$ `LIS(4)` $\rightarrow$ `LIS(3)` $\rightarrow$ `LIS(0)`, so the optimal solution (read backwards) is index {0, 3, 4, 5} or {-7, 2, 3, 8}.

The LIS problem can also be solved using the *output-sensitive* $O(n \log k)$ greedy + D&C algorithm (where k is the length of the LIS) instead of $O(n^2)$ by maintaining an array that is *always sorted* and therefore amenable to binary search. Let array `L` be an array such that `L(i)` represents the smallest ending value of all length-`i` LISs found so far. Though this definition is slightly complicated, it is easy to see that it is always ordered—`L(i-1)` will always be smaller than `L(i)` as the second-last element of any LIS (of length-`i`) is smaller than its last element. As such, we can binary search array `L` to determine the longest possible subsequence we can create by appending the current element `A[i]`—simply find the index of the last element in `L` that is less than `A[i]`. Using the same example, we will update array `L` step by step using this algorithm:

- Initially, at `A[0]` = -7, we have `L` = {-7}.
- We can insert `A[1]` = 10 at `L[1]` so that we have a length-2 LIS, `L` = {-7, <u>10</u>}.
- For `A[2]` = 9, we replace `L[1]` so that we have a 'better' length-2 LIS ending:
  `L` = {-7, <u>9</u>}.
  This is a *greedy* strategy. By storing the LIS with smaller ending value, we maximize our ability to further extend the LIS with future values.
- For `A[3]` = 2, we replace `L[1]` to get an 'even better' length-2 LIS ending:
  `L` = {-7, <u>2</u>}.
- We insert `A[4]` = 3 at `L[2]` so that we have a longer LIS, `L` = {-7, 2, <u>3</u>}.
- We insert `A[5]` = 8 at `L[3]` so that we have a longer LIS, `L` = {-7, 2, 3, <u>8</u>}.
- For `A[6]` = 8, nothing changes as `L[3]` = 8.
  `L` = {-7, 2, 3, 8} remains unchanged.
- For `A[7]` = 1, we improve `L[1]` so that `L` = {-7, <u>1</u>, 3, 8}.
  This illustrates how the array `L` is *not* the LIS of `A`. This step is important as there can be longer subsequences *in the future* that may extend the length-2 subsequence at `L[1]` = 1. For example, try this test case: `A` = {<u>-7</u>, 10, 9, 2, 3, 8, 8, <u>1, 2, 3, 4</u>}. The length of LIS for this test case is 5.
- The answer is the largest length of the sorted array `L` at the end of the process.

Source code: `ch3_06_LIS.cpp/java`

---

4. 0-1 Knapsack (Subset Sum)

Problem[14]: Given n items, each with its own value V_i and weight W_i, $\forall$`i` $\in$ `[0..n-1]`, and a maximum knapsack size S, compute the maximum value of the items that we can carry, if we can either[15] ignore or take a particular item (hence the term 0-1 for ignore/take).

---

[14]This problem is also known as the Subset Sum problem. It has a similar problem description: Given a set of integers and an integer S, is there a (non-empty) subset that has a sum equal to S?

[15]There are other variants of this problem, e.g. the Fractional Knapsack problem with Greedy solution.

Example: $n = 4$, $V = \{100, 70, 50, 10\}$, $W = \{10, 4, 6, 12\}$, $S = 12$.
If we select item 0 with weight 10 and value 100, we cannot take any other item. Not optimal.
If we select item 3 with weight 12 and value 10, we cannot take any other item. Not optimal.
If we select item 1 and 2, we have total weight 10 and total value 120. This is the maximum.

Solution: Use these Complete Search recurrences `val(id, remW)` where `id` is the index of the current item to be considered and `remW` is the remaining weight left in the knapsack:

1. `val(id, 0) = 0` // if `remW = 0`, we cannot take anything else
2. `val(n, remW) = 0` // if `id = n`, we have considered all items
3. if `W[id] > remW`, we have no choice but to ignore this item
`val(id, remW) = val(id + 1, remW)`
4. if `W[id] ≤ remW`, we have two choices: ignore or take this item; we take the maximum
`val(id, remW) = max(val(id + 1, remW), V[id] + val(id + 1, remW - W[id]))`

The answer can be found by calling `value(0, S)`. Note the overlapping sub-problems in this 0-1 Knapsack problem. Example: After taking item 0 and ignoring item 1-2, we arrive at state `(3, 2)`—at the third item (`id = 3`) with two units of weight left (`remW = 2`). After ignoring item 0 and taking item 1-2, we also arrive at the same state `(3, 2)`. Although there are overlapping sub-problems, there are only $O(nS)$ possible distinct states (as `id` can vary between `[0..n-1]` and `remW` can vary between `[0..S]`)! We can compute each of these states in $O(1)$, thus the overall time complexity[16] of this DP solution is $O(nS)$.

Note: The top-down version of this DP solution is often faster than the bottom-up version. This is because not all states are actually visited, and hence the critical DP states involved are actually only a (very small) subset of the entire state space. Remember: The top-down DP only visits *the required states* whereas bottom-up DP visits *all distinct states*. Both versions are provided in our source code library.

Source code: `ch3_07_UVa10130.cpp/java`

5. Coin Change (CC) - The General Version

Problem: Given a target amount V cents and a list of denominations for n coins, i.e. we have `coinValue[i]` (in cents) for coin types i ∈ `[0..n-1]`, what is the minimum number of coins that we must use to represent V? Assume that we have unlimited supply of coins of any type (also see Section 3.4.1).

Example 1: $V = 10$, $n = 2$, `coinValue = {1, 5}`; We can use:
A. Ten 1 cent coins = $10 \times 1 = 10$; Total coins used = 10
B. One 5 cents coin + Five 1 cent coins = $1 \times 5 + 5 \times 1 = 10$; Total coins used = 6
C. Two 5 cents coins = $2 \times 5 = 10$; Total coins used = $2 \rightarrow$ Optimal

We can use the Greedy algorithm if the coin denominations are suitable (see Section 3.4.1). Example 1 above is solvable with the Greedy algorithm. However, for general cases, we have to use DP. See Example 2 below:

Example 2: $V = 7$, $n = 4$, `coinValue = {1, 3, 4, 5}`
The Greedy approach will produce 3 coins as its result as $5+1+1 = 7$, but the optimal solution is actually 2 coins (from 4+3)!

Solution: Use these Complete Search recurrence relations for `change(value)`, where `value` is the remaining amount of cents that we need to represent in coins:

---

[16]If S is large such that $NS >> 1M$, this DP solution is not feasible, even with the space saving trick!

1. `change(0)` = 0 // we need 0 coins to produce 0 cents
2. `change(< 0)` = ∞ // in practice, we can return a large positive value
3. `change(value)` = 1 + min(change(value - coinValue[i])) $\forall i \in$ [0..n-1]

The answer can be found in the return value of `change(V)`.

<0	0	1	2	3	4	5	6	7	8	9	10
∞	0	1	2	3	4	1	2	3	4	5	2

V = 10, N = 2, coinValue= {1, 5}

Figure 3.10: Coin Change

Figure 4.2.3 shows that:
`change(0)` = 0 and `change(< 0)` = ∞: These are the base cases.
`change(1)` = 1, from 1 + `change(1-1)`, as 1 + `change(1-5)` is infeasible (returns ∞).
`change(2)` = 2, from 1 + `change(2-1)`, as 1 + `change(2-5)` is also infeasible (returns ∞).
... same thing for `change(3)` and `change(4)`.
`change(5)` = 1, from 1 + `change(5-5)` = 1 coin, smaller than 1 + `change(5-1)` = 5 coins.
... and so on until `change(10)`.
The answer is in `change(V)`, which is `change(10)` = 2 in this example.

We can see that there are a lot of overlapping sub-problems in this Coin Change problem (e.g. both `change(10)` and `change(6)` require the value of `change(5)`). However, there are only $O(V)$ possible distinct states (as `value` can vary between [0..V])! As we need to try n types of coins per state, the overall time complexity of this DP solution is $O(nV)$.

A variant of this problem is to count *the number of possible (canonical) ways* to get value V cents using a list of denominations of n coins. For example 1 above, the answer is 3: {1+1+1+1+1 + 1+1+1+1+1, 5 + 1+1+1+1+1, 5 + 5}.

Solution: Use these Complete Search recurrence relation: `ways(type, value)`, where `value` is the same as above but we now have one more parameter `type` for the index of the coin type that we are currently considering. This second parameter `type` is important as this solution considers the coin types sequentially. Once we choose to ignore a certain coin type, we should not consider it again to avoid double-counting:

1. `ways(type, 0)` = 1 // one way, use nothing
2. `ways(type, <0)` = 0 // no way, we cannot reach negative value
3. `ways(n, value)` = 0 // no way, we have considered all coin types $\in$ [0..n-1]
4. `ways(type, value)` = `ways(type + 1, value)` + // if we ignore this coin type,
`ways(type, value - coinValue[type])` // plus if we use this coin type

There are only $O(nV)$ possible distinct states. Since each state can be computed in $O(1)$, the overall time complexity[17] of this DP solution is $O(nV)$. The answer can be found by calling `ways(0, V)`. Note: If the coin values are not changed and you are given many queries with different `V`, then we can choose *not* to reset the memo table. Therefore, we run this $O(nV)$ algorithm once and just perform an $O(1)$ lookup for subsequent queries.

Source code (this coin change variant): `ch3_08_UVa674.cpp/java`

---

[17]If V is large such that $nV \gg 1M$, this DP solution is not feasible even with the space saving trick!

6. Traveling Salesman Problem (TSP)

Problem: Given n cities and their pairwise distances in the form of a matrix `dist` of size $n \times n$, compute the cost of making a tour[18] that starts from any city s, goes through all the other $n - 1$ cities *exactly once*, and finally returns to the starting city s.

Example: The graph shown in Figure 3.11 has $n = 4$ cities. Therefore, we have $4! = 24$ possible tours (permutations of 4 cities). One of the minimum tours is `A-B-C-D-A` with a cost of `20+30+12+35 = 97` (notice that there can be more than one optimal solution).
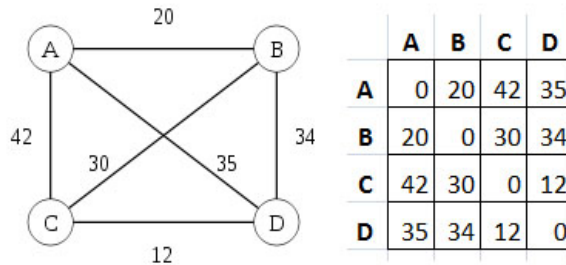
Figure 3.11: A Complete Graph

A 'brute force' TSP solution (either iterative or recursive) that tries all $O((n - 1)!)$ possible tours (fixing the first city to vertex A in order to take advantage of symmetry) is only effective when n is at most 12 as $11! \approx 40M$. When $n > 12$, such brute force solutions will get a TLE in programming contests. However, if there are multiple test cases, the limit for such 'brute force' TSP solution is probably just $n = 11$.

We can utilize DP for TSP since the computation of sub-tours is clearly overlapping, e.g. the tour $A - B - C - (n - 3)$ *other cities that finally return to A* clearly overlaps the tour $A - C - B -$*the same* $(n - 3)$ *other cities that also return to A*. If we can avoid re-computing the lengths of such sub-tours, we can save a lot of computation time. However, a distinct state in TSP depends on two parameters: The last city/vertex visited `pos` and something that we may have not seen before—a *subset* of visited cities.

There are many ways to represent a set. However, since we are going to pass this set information around as a parameter of a recursive function (if using top-down DP), the representation we use must be lightweight and efficient! In Section 2.2, we have presented a viable option for this usage: The *bitmask*. If we have n cities, we use a binary integer of length n. If bit i is '1' (on), we say that item (city) i is inside the set (it has been visited) and item i is not inside the set (and has not been visited) if the bit is instead '0' (off). For example: `mask`$= 18_{10} = 10010_2$ implies that items (cities) $\{1, 4\}$ are in[19] the set (and have been visited). Recall that to check if bit i is on or off, we can use `mask & (1 << i)`. To set bit i, we can use `mask |= (1 << i)`.

Solution: Use these Complete Search recurrence relations for `tsp(pos, mask)`:

1. `tsp(pos, `$2^n - 1$`) = dist[pos][0]` // all cities have been visited, return to starting city
// Note: `mask = (1 << n) - 1` or $2^n - 1$ implies that all n bits in `mask` are on.
2. `tsp(pos, mask) = min(dist[pos][nxt] + tsp(nxt, mask | (1 << nxt)))`
// $\forall$ `nxt` $\in$ `[0..n-1]`, `nxt != pos`, and `(mask & (1 << nxt))` is '0' (turned off)
// We basically tries all possible next cities that have not been visited before at each step.

There are only $O(n \times 2^n)$ distinct states because there are n cities and we remember up to 2^n other cities that have been visited in each tour. Each state can be computed in $O(n)$,

---

[18]Such a tour is called a Hamiltonian tour, which is a cycle in an undirected graph which visits each vertex exactly once and also returns to the starting vertex.

[19]Remember that in `mask`, indices starts from 0 and are counted from the right.

thus the overall time complexity of this DP solution is $O(2^n \times n^2)$. This allows us to solve up to[20] $n \approx 16$ as $16^2 \times 2^{16} \approx 17M$. This is not a huge improvement over the brute force solution but if the programming contest problem involving TSP has input size $11 \le n \le 16$, then DP is the solution, not brute force. The answer can be found by calling `tsp(0, 1)`: We start from city 0 (we can start from any vertex; but the simplest choice is vertex 0) and set `mask = 1` so that city 0 is never re-visited again.

Usually, DP TSP problems in programming contests require some kind of graph preprocessing to generate the distance matrix `dist` before running the DP solution. These variants are discussed in Section 8.4.3.

DP solutions that involve a (small) set of Booleans as one of the parameters are more well known as the DP with bitmask technique. More challenging DP problems involving this technique are discussed in Section 8.3 and 9.2.

Visualization: `www.comp.nus.edu.sg/~stevenha/visualization/rectree.html`

Source code: `ch3_09_UVa10496.cpp/java`

Exercise 3.5.2.1: The solution for the Max 2D Range Sum problem runs in $O(n^4)$. Actually, there exists an $O(n^3)$ solution that combines the DP solution for the Max Range 1D Sum problem on one dimension and uses the same idea as proposed by Kadane on the other dimension. Solve UVa 108 with an $O(n^3)$ solution!

Exercise 3.5.2.2: The solution for the `Range Minimum Query(i, j)` on 1D arrays in Section 2.4.3 uses Segment Tree. This is overkill if the given array is static and unchanged throughout all the queries. Use a DP technique to answer `RMQ(i, j)` in $O(n \log n)$ preprocessing and $O(1)$ per query.

Exercise 3.5.2.3: Solve the LIS problem using the $O(n \log k)$ solution and *also* reconstruct one of the LIS.

Exercise 3.5.2.4: Can we use an iterative Complete Search technique that tries all possible subsets of n items as discussed in Section 3.2.1 to solve the 0-1 Knapsack problem? What are the limitations, if any?

Exercise 3.5.2.5*: Suppose we add one more parameter to this classic 0-1 Knapsack problem. Let K_i denote the number of copies of item i for use in the problem. Example: $n = 2$, $V = \{100, 70\}$, $W = \{5, 4\}$, $K = \{2, 3\}$, $S = 17$ means that there are two copies of item 0 with weight 5 and value 100 and there are three copies of item 1 with weight 4 and value 70. The optimal solution for this example is to take one of item 0 and three of item 1, with a total weight of 17 and total value 310. Solve new variant of the problem assuming that $1 \le n \le 500$, $1 \le S \le 2000$, $n \le \sum_{i=0}^{n-1} K_i \le 40000$! Hint: Every integer can be written as a sum of powers of 2.

Exercise 3.5.2.6*: The DP TSP solution shown in this section can still be *slightly* enhanced to make it able to solve test case with $n = 17$ in contest environment. Show the required minor change to make this possible! Hint: Consider symmetry!

Exercise 3.5.2.7*: On top of the minor change asked in **Exercise 3.5.2.5***, what *other change(s)* is/are needed to have a DP TSP solution that is able to handle $n = 18$ (or even $n = 19$, but with much lesser number of test cases)?

---

[20]As programming contest problems usually require exact solutions, the DP-TSP solution presented here is already one of the best solutions. In real life, the TSP often needs to be solved for instances with thousands of cities. To solve larger problems like that, we have non-exact approaches like the ones presented in [26].

3.5.3 Non-Classical Examples

Although DP is the single most popular problem type with the highest frequency of appearance in recent programming contests, the classical DP problems in their *pure forms* usually never appear in modern ICPCs or IOIs again. We study them to understand DP, but we have to learn to solve many other non-classical DP problems (which may become classic in the near future) and develop our 'DP skills' in the process. In this subsection, we discuss two more non-classical examples, adding to the UVa 11450 - Wedding Shopping problem that we have discussed in detail earlier. We have also selected some easier non-classical DP problems as programming exercises. Once you have cleared most of these problems, you are welcome to explore the more challenging ones in the other sections in this book, e.g. Section 4.7.1, 5.4, 5.6, 6.5, 8.3, 9.2, 9.21, etc.

1. UVa 10943 - How do you add?

Abridged problem description: Given an integer n, how many ways can K non-negative integers less than or equal to n add up to n? Constraints: $1 \leq n, K \leq 100$. Example: For $n = 20$ and $K = 2$, there are 21 ways: $0 + 20$, $1 + 19$, $2 + 18$, $3 + 17$, ..., $20 + 0$.

Mathematically, the number of ways can be expressed as $^{(n+k-1)}C_{(k-1)}$ (see Section 5.4.2 about Binomial Coefficients). We will use this simple problem to re-illustrate Dynamic Programming principles that we have discussed in this section, especially the process of deriving appropriate states for a problem and deriving correct transitions from one state to another given the base case(s).

First, we have to determine the parameters of this problem to be selected to represent distinct states of this problem. There are only two parameters in this problem, n and K. Therefore, there are only 4 possible combinations:

1. If we do not choose any of them, we cannot represent a state. This option is ignored.

2. If we choose only n, then we do not know how many numbers $\leq n$ have been used.

3. If we choose only K, then we do not know the target sum n.

4. Therefore, the state of this problem should be represented by a pair (or tuple) (n, K). The order of chosen parameter(s) does not matter, i.e. the pair (K, n) is also OK.

Next, we have to determine the base case(s). It turns out that this problem is very easy when $K = 1$. Whatever n is, there is only *one way* to add exactly one number less than or equal to n to get n: Use n itself. There is no other base case for this problem.

For the general case, we have this recursive formulation which is not too difficult to derive: At state (n, K) where $K > 1$, we can split n into one number $X \in$ [0..n] and $n - X$, i.e. $n = X + (n - X)$. By doing this, we arrive at the subproblem $(n - X, K - 1)$, i.e. given a number $n - X$, how many ways can $K - 1$ numbers less than or equal to $n - X$ add up to $n - X$? We can then sum all these ways.

These ideas can be written as the following Complete Search recurrence `ways(n, K)`:

1. `ways(n, 1) = 1` // we can only use 1 number to add up to n, the number n itself

2. `ways(n, K) = ` $\sum_{X=0}^{n}$ `ways(n - X, K - 1)` // sum all possible ways, recursively

This problem has overlapping sub-problems. For example, the test case $n = 1, K = 3$ has overlapping sub-problems: The state $(n = 0, K = 1)$ is reached twice (see Figure 4.39 in Section 4.7.1). However, there are only $n \times K$ possible states of (n, K). The cost of computing each state is $O(n)$. Thus, the overall time complexity is $O(n^2 \times K)$. As $1 \leq n, K \leq 100$, this is feasible. The answer can be found by calling `ways(n, K)`.

Note that this problem actually just needs the result modulo $1M$ (i.e. the last 6 digits of the answer). See Section 5.5.8 for a discussion on modulo arithmetic computation.

Source code: `ch3_10_UVa10943.cpp/java`

2. UVa 10003 - Cutting Sticks

Abridged problem statement: Given a stick of length $1 \le l \le 1000$ and $1 \le n \le 50$ cuts to be made to the stick (the cut coordinates, lying in the range [0..l], are given). The cost of a cut is determined by the length of the stick to be cut. Your task is to find a cutting sequence so that the overall cost is minimized.

Example: $l = 100$, $n = 3$, and cut coordinates: `A` $= \{25, 50, 75\}$ (already sorted)

If we cut from left to right, then we will incur cost $= 225$.
1. First cut is at coordinate 25, total cost so far $= 100$;
2. Second cut is at coordinate 50, total cost so far $= 100 + 75 = 175$;
3. Third cut is at coordinate 75, final total cost $= 175 + 50 = 225$;
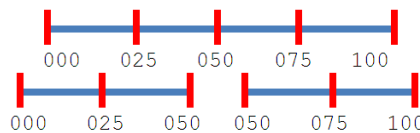
Figure 3.12: Cutting Sticks Illustration

However, the optimal answer is 200.
1. First cut is at coordinate 50, total cost so far $= 100$; (this cut is shown in Figure 3.12)
2. Second cut is at coordinate 25, total cost so far $= 100 + 50 = 150$;
3. Third cut is at coordinate 75, final total cost $= 150 + 50 = 200$;

How do we tackle this problem? An initial approach might be this Complete Search algorithm: Try all possible cutting points. Before that, we have to select an appropriate state definition for the problem: The (intermediate) sticks. We can describe a stick with its two endpoints: `left` and `right`. However, these two values can be very huge and this can complicate the solution later when we want to memoize their values. We can take advantage of the fact that there are only $n + 1$ smaller sticks after cutting the original stick n times. The endpoints of each smaller stick can be described by 0, the cutting point coordinates, and l. Therefore, we will add two more coordinates so that `A` $= \{0$, the original `A`, and $l\}$ so that we can denote a stick by the indices of its endpoints in `A`.
　　We can then use these recurrences for `cut(left, right)`, where `left/right` are the left/right indices of the current stick w.r.t. `A`. Originally, the stick is described by `left = 0` and `right` $= n+1$, i.e. a stick with length [0..l]:

1. `cut(i-1, i) = 0`, $\forall i \in$ [1..$n+1$] // if `left + 1 = right` where `left` and `right` are the indices in `A`, then we have a stick segment that does not need to be divided further.

2. `cut(left, right) = min(cut(left, i) + cut(i, right) + (A[right]-A[left]))`
$\forall i \in$ [`left+1..right-1`] // try all possible cutting points and pick the best.
The cost of a cut is the length of the current stick, captured in (`A[right]-A[left]`).
The answer can be found at `cut(0, `$n+1$`)`.

Now let's analyze the time complexity. Initially, we have n choices for the cutting points. Once we cut at a certain cutting point, we are left with $n - 1$ further choices of the second

cutting point. This repeats until we are left with zero cutting points. Trying all possible cutting points this way leads to an $O(n!)$ algorithm, which is impossible for $1 \leq n \leq 50$.

However, this problem has overlapping sub-problems. For example, in Figure 3.12 above, cutting at index 2 (cutting point = 50) produces two states: (0, 2) and (2, 4). The same state (2, 4) can also be reached by cutting at index 1 (cutting point 25) and then cutting at index 2 (cutting point 50). Thus, the search space is actually not that large. There are only $(n + 2) \times (n + 2)$ possible left/right indices or $O(n^2)$ distinct states and be memoized. The time to required to compute one state is $O(n)$. Thus, the overall time complexity (of the top-down DP) is $O(n^3)$. As $n \leq 50$, this is a feasible solution.

Source code: ch3_11_UVa10003.cpp/java

Exercise 3.5.3.1*: Almost all of the source code shown in this section (LIS, Coin Change, TSP, and UVa 10003 - Cutting Sticks) are written in a top-down DP fashion due to the preferences of the authors of this book. Rewrite them using the bottom-up DP approach.

Exercise 3.5.3.2*: Solve the Cutting Sticks problem in $O(n^2)$. Hint: Use Knuth-Yao DP Speedup by utilizing that the recurrence satisfies Quadrangle Inequality (see [2]).

Remarks About Dynamic Programming in Programming Contests

Basic (Greedy and) DP techniques techniques are always included in popular algorithm textbooks, e.g. Introduction to Algorithms [7], Algorithm Design [38] and Algorithm [8]. In this section, we have discussed six classical DP problems and their solutions. A brief summary is shown in Table 3.4. These classical DP problems, if they are to appear in a programming contest today, will likely occur only as part of bigger and harder problems.

	1D RSQ	2D RSQ	LIS	Knapsack	CC	TSP
State	(i)	(i,j)	(i)	(id,remW)	(v)	(pos,mask)
Space	$O(n)$	$O(n^2)$	$O(n)$	$O(nS)$	$O(V)$	$O(n2^n)$
Transition	subarray	submatrix	all $j < i$	take/ignore	all n coins	all n cities
Time	$O(1)$	$O(1)$	$O(n^2)$	$O(nS)$	$O(nV)$	$O(2^n n^2)$

Table 3.4: Summary of Classical DP Problems in this Section

To help keep up with the growing difficulty and creativity required in these techniques (especially the non-classical DP), we recommend that you also read the TopCoder algorithm tutorials [30] and attempt the more recent programming contest problems.

In this book, we will revisit DP again on several occasions: Floyd Warshall's DP algorithm (Section 4.5), DP on (implicit) DAG (Section 4.7.1), String Alignment (Edit Distance), Longest Common Subsequence (LCS), other DP on String algorithms (Section 6.5), More Advanced DP (Section 8.3), and several topics on DP in Chapter 9.

In the past (1990s), a contestant who is good at DP can become a 'king of programming contests' as DP problems were usually the 'decider problems'. Now, mastering DP is a *basic* requirement! You cannot do well in programming contests without this knowledge. However, we have to keep reminding the readers of this book not to claim that they know DP if they only memorize the solutions of the classical DP problems! Try to master the art of DP problem solving: Learn to determine the states (the DP table) that can uniquely

and efficiently represent sub-problems and also how to fill up that table, either via top-down recursion or bottom-up iteration.

There is no better way to master these problem solving paradigms than solving real programming problems! Here, we list several examples. Once you are familiar with the examples shown in this section, study the newer DP problems that have begun to appear in recent programming contests.

Programming Exercises solvable using Dynamic Programming:

- Max 1D Range Sum
    1. UVa 00507 - Jill Rides Again (standard problem)
    2. **UVa 00787 - Maximum Sub ... *** (max 1D range *product*, be careful with 0, use Java BigInteger, see Section 5.3)
    3. **UVa 10684 - The Jackpot *** (standard problem; easily solvable with the given sample source code)
    4. **UVa 10755 - Garbage Heap *** (combination of max 2D range sum in two of the three dimensions—see below—and max 1D range sum using Kadane's algorithm on the third dimension)
    See more examples in Section 8.4.

- Max 2D Range Sum
    1. **UVa 00108 - Maximum Sum *** (discussed in this section with sample source code)
    2. UVa 00836 - Largest Submatrix (convert '0' to -INF)
    3. UVa 00983 - Localized Summing for ... (max 2D range sum, get submatrix)
    4. UVa 10074 - Take the Land (standard problem)
    5. UVa 10667 - Largest Block (standard problem)
    6. **UVa 10827 - Maximum Sum on ... *** (copy $n \times n$ matrix into $n \times 2n$ matrix; then this problem becomes a standard problem again)
    7. **UVa 11951 - Area *** (use long long; max 2D range sum; prune the search space whenever possible)

- Longest Increasing Subsequence (LIS)
    1. UVa 00111 - History Grading (be careful of the ranking system)
    2. UVa 00231 - Testing the Catcher (straight-forward)
    3. UVa 00437 - The Tower of Babylon (can be modeled as LIS)
    4. **UVa 00481 - What Goes Up? *** (use $O(n \log k)$ LIS; print solution; see our sample source code)
    5. UVa 00497 - Strategic Defense Initiative (solution must be printed)
    6. UVa 01196 - Tiling Up Blocks (LA 2815, Kaohsiung03; sort all the blocks in increasing `L[i]`, then we get the classical LIS problem)
    7. UVa 10131 - Is Bigger Smarter? (sort elephants based on decreasing IQ; LIS on increasing weight)
    8. UVa 10534 - Wavio Sequence (must use $O(n \log k)$ LIS twice)
    9. *UVa 11368 - Nested Dolls* (sort in one dimension, LIS in the other)
    10. **UVa 11456 - Trainsorting *** (max(LIS(i) + LDS(i) - 1), $\forall i \in [0 \dots n\text{-}1]$)
    11. **UVa 11790 - Murcia's Skyline *** (combination of LIS+LDS, weighted)

- 0-1 Knapsack (Subset Sum)

  1. UVa 00562 - Dividing Coins (use a one dimensional table)
  2. UVa 00990 - Diving For Gold (print the solution)
  3. UVa 01213 - Sum of Different Primes (LA 3619, Yokohama06, extension of 0-1 Knapsack, use three parameters: (id, remN, remK) on top of (id, remN))
  4. UVa 10130 - SuperSale (discussed in this section with sample source code)
  5. UVa 10261 - Ferry Loading (s: current car, left, right)
  6. **UVa 10616 - Divisible Group Sum *** (input can be -ve, use long long)
  7. UVa 10664 - Luggage (Subset Sum)
  8. **UVa 10819 - Trouble of 13-Dots *** (0-1 knapsack with 'credit card' twist!)
  9. *UVa 11003 - Boxes* (try all max weight from 0 to $max(weight[i]+capacity[i])$, $\forall i \in [0..n\text{-}1]$; if a max weight is known, how many boxes can be stacked?)
  10. UVa 11341 - Term Strategy (s: id, h_learned, h_left; t: learn module 'id' by 1 hour or skip)
  11. ***UVa 11566 - Let's Yum Cha **** (English reading problem, actually just a knapsack variant: double each dim sum and add one parameter to check if we have bought too many dishes)
  12. UVa 11658 - Best Coalition (s: id, share; t: form/ignore coalition with id)

- Coin Change (CC)

  1. UVa 00147 - Dollars (similar to UVa 357 and UVa 674)
  2. UVa 00166 - Making Change (two coin change variants in one problem)
  3. **UVa 00357 - Let Me Count The Ways *** (similar to UVa 147/674)
  4. UVa 00674 - Coin Change (discussed in this section with sample source code)
  5. **UVa 10306 - e-Coins *** (variant: each coin has two components)
  6. UVa 10313 - Pay the Price (modified coin change + DP 1D range sum)
  7. UVa 11137 - Ingenuous Cubrency (use long long)
  8. **UVa 11517 - Exact Change *** (a variation to the coin change problem)

- Traveling Salesman Problem (TSP)

  1. **UVa 00216 - Getting in Line *** (TSP, still solvable with backtracking)
  2. **UVa 10496 - Collecting Beepers *** (discussed in this section with sample source code; actually, since $n \leq 11$, this problem is still solvable with recursive backtracking and sufficient pruning)
  3. **UVa 11284 - Shopping Trip *** (requires shortest paths pre-processing; TSP variant where we can go home early; we just need to tweak the DP TSP recurrence a bit: at each state, we have one more option: go home early)
     See more examples in Section 8.4.3 and Section 9.2.

- Non Classical (The Easier Ones)

  1. UVa 00116 - Unidirectional TSP (similar to UVa 10337)
  2. *UVa 00196 - Spreadsheet* (notice that the dependencies of cells are acyclic; we can therefore memoize the direct (or indirect) value of each cell)
  3. *UVa 01261 - String Popping* (LA 4844, Daejeon10, a simple backtracking problem; but we use a set<string> to prevent the same state (a substring) from being checked twice)
  4. UVa 10003 - Cutting Sticks (discussed in details in this section with sample source code)
  5. UVa 10036 - Divisibility (must use offset technique as value can be negative)

6. *UVa 10086 - Test the Rods* (s: idx, rem1, rem2; which site that we are now, up to 30 sites; remaining rods to be tested at NCPC; and remaining rods to be tested at BCEW; t: for each site, we split the rods, x rods to be tested at NCPC and $m[i] - x$ rods to be tested at BCEW; print the solution)

7. **UVa 10337 - Flight Planner *** (DP; shortest paths on DAG)

8. UVa 10400 - Game Show Math (backtracking with clever pruning is sufficient)

9. *UVa 10446 - The Marriage Interview* (edit the given recursive function a bit, add memoization)

10. UVa 10465 - Homer Simpson (one dimensional DP table)

11. *UVa 10520 - Determine it* (just write the given formula as a top-down DP with memoization)

12. *UVa 10688 - The Poor Giant* (note that the sample in the problem description is a bit wrong, it should be: $1+(1+3)+(1+3)+(1+3) = 1+4+4+4 = 13$, beating 14; otherwise a simple DP)

13. **UVa 10721 - Bar Codes *** (s: n, k; t: try all from 1 to m)

14. UVa 10910 - Mark's Distribution (two dimensional DP table)

15. UVa 10912 - Simple Minded Hashing (s: len, last, sum; t: try next char)

16. **UVa 10943 - How do you add? *** (discussed in this section with sample source code; s: n, k; t: try all the possible splitting points; alternative solution is to use the closed form mathematical formula: $C(n + k - 1, k - 1)$ which also needs DP, see Section 5.4)

17. *UVa 10980 - Lowest Price in Town* (simple)

18. *UVa 11026 - A Grouping Problem* (DP, similar idea with binomial theorem in Section 5.4)

19. UVa 11407 - Squares (can be memoized)

20. UVa 11420 - Chest of Drawers (s: prev, id, numlck; lock/unlock this chest)

21. UVa 11450 - Wedding Shopping (discussed in details in this section with sample source code)

22. UVa 11703 - sqrt log sin (can be memoized)

- Other Classical DP Problems in this Book

  1. Floyd Warshall's for All-Pairs Shortest Paths problem (see Section 4.5)
  2. String Alignment (Edit Distance) (see Section 6.5)
  3. Longest Common Subsequence (see Section 6.5)
  4. Matrix Chain Multiplication (see Section 9.20)
  5. Max (Weighted) Independent Set (on tree, see Section 9.22)

- Also see Section 4.7.1, 5.4, 5.6, 6.5, 8.3, 8.4 and parts of Chapter 9 for *more* programming exercises related to Dynamic Programming.

3.6 Solution to Non-Starred Exercises

Exercise 3.2.1.1: This is to avoid the division operator so that we only work with integers! If we iterate through `abcde` instead, we may encounter a non-integer result when we compute `fghij = abcde / N`.

Exercise 3.2.1.2: It wil get an AC too as $10! \approx 3$ million, about the same as the algorithm presented in Section 3.2.1.

Exercise 3.2.2.1: Modify the backtrack function to resemble this code:

```
void backtrack(int c) {
  if (c == 8 && row[b] == a) {          // candidate sol, (a, b) has 1 queen
    printf("%2d      %d", ++lineCounter, row[0] + 1);
    for (int j = 1; j < 8; j++) printf(" %d", row[j] + 1);
    printf("\n"); }
  for (int r = 0; r < 8; r++)                        // try all possible row
    if (col == b && r != a) continue;                        // ADD THIS LINE
    if (place(r, c)) {          // if can place a queen at this col and row
      row[c] = r; backtrack(c + 1);      // put this queen here and recurse
}   }
```

Exercise 3.3.1.1: This problem can be solved without the 'binary search the answer' technique. Simulate the journey once. We just need to find the largest fuel requirement in the entire journey and make the fuel tank be sufficient for it.

Exercise 3.5.1.1: Garment $g = 0$, take the third model (cost 8); Garment $g = 1$, take the first model (cost 10); Garment $g = 2$, take the first model (cost 7); Money used = 25. Nothing left. Test case C is also solvable with Greedy algorithm.

Exercise 3.5.1.2: No, this state formulation does not work. We need to know how much money we have left at each sub-problem so that we can determine if we still have enough money to buy a certain model of the current garment.

Exercise 3.5.1.3: The modified bottom-up DP code is shown below:

```
#include <cstdio>
#include <cstring>
using namespace std;

int main() {
  int g, money, k, TC, M, C, cur;
  int price[25][25];
  bool reachable[2][210]; // reachable table[ONLY TWO ROWS][money (<= 200)]
  scanf("%d", &TC);
  while (TC--) {
    scanf("%d %d", &M, &C);
    for (g = 0; g < C; g++) {
      scanf("%d", &price[g][0]);
      for (money = 1; money <= price[g][0]; money++)
        scanf("%d", &price[g][money]);
    }
```

```
      memset(reachable, false, sizeof reachable);
      for (g = 1; g <= price[0][0]; g++)
        if (M - price[0][g] >= 0)
          reachable[0][M - price[0][g]] = true;

      cur = 1;                                       // we start with this row
      for (g = 1; g < C; g++) {
        memset(reachable[cur], false, sizeof reachable[cur]);    // reset row
        for (money = 0; money < M; money++) if (reachable[!cur][money])
          for (k = 1; k <= price[g][0]; k++) if (money - price[g][k] >= 0)
            reachable[cur][money - price[g][k]] = true;
        cur = !cur;                       // IMPORTANT TRICK: flip the two rows
      }

      for (money = 0; money <= M && !reachable[!cur][money]; money++);

      if (money == M + 1) printf("no solution\n");  // last row has no on bit
      else                printf("%d\n", M - money);
} } // return 0;
```

Exercise 3.5.2.1: The $O(n^3)$ solution for Max 2D Range Sum problem is shown below:

```
scanf("%d", &n);                        // the dimension of input square matrix
for (int i = 0; i < n; i++) for (int j = 0; j < n; j++) {
  scanf("%d", &A[i][j]);
  if (j > 0) A[i][j] += A[i][j - 1];       // only add columns of this row i
}

maxSubRect = -127*100*100;     // the lowest possible value for this problem
for (int l = 0; l < n; l++) for (int r = l; r < n; r++) {
  subRect = 0;
  for (int row = 0; row < n; row++) {
    // Max 1D Range Sum on columns of this row i
    if (l > 0) subRect += A[row][r] - A[row][l - 1];
    else       subRect += A[row][r];

    // Kadane's algorithm on rows
    if (subRect < 0) subRect = 0;       // greedy, restart if running sum < 0
    maxSubRect = max(maxSubRect, subRect);
  }
}
```

Exercise 3.5.2.2: The solution is given in Section 9.33.

Exercise 3.5.2.3: The solution is already written inside ch3_06_LIS.cpp/java.

Exercise 3.5.2.4: The iterative Complete Search solution to generate and check all possible subsets of size n runs in $O(n \times 2^n)$. This is OK for $n \le 20$ but too slow when $n > 20$. The DP solution presented in Section 3.5.2 runs in $O(n \times S)$. If S is not that large, we can have a much larger n than just 20 items.

3.7 Chapter Notes

Many problems in ICPC or IOI require a combination (see Section 8.4) of these problem solving strategies. If we have to nominate only one chapter in this book that contestants have to really master, we would choose this one.

In Table 3.5, we compare the four problem solving techniques in their likely results for various problem types. In Table 3.5 and the list of programming exercises in this section, you will see that there are *many more* Complete Search and DP problems than D&C and Greedy problems. Therefore, we recommend that readers concentrate on improving their Complete Search and DP skills.

	BF Problem	D&C Problem	Greedy Problem	DP Problem
BF Solution	AC	TLE/AC	TLE/AC	TLE/AC
D&C Solution	WA	AC	WA	WA
Greedy Solution	WA	WA	AC	WA
DP Solution	MLE/TLE/AC	MLE/TLE/AC	MLE/TLE/AC	AC
Frequency	High	(Very) Low	Low	High

Table 3.5: Comparison of Problem Solving Techniques (Rule of Thumb only)

We will conclude this chapter by remarking that for some real-life problems, especially those that are classified as NP-hard [7], many of the approaches discussed in this section will not work. For example, the 0-1 Knapsack Problem which has an $O(nS)$ DP complexity is too slow if S is big; TSP which has a $O(2^n \times n^2)$ DP complexity is too slow if n is any larger than 18 (see **Exercise 3.5.2.7***). For such problems, we can resort to heuristics or local search techniques such as Tabu Search [26, 25], Genetic Algorithms, Ant-Colony Optimizations, Simulated Annealing, Beam Search, etc. However, all these heuristic-based searches are not in the IOI syllabus [20] and also not widely used in ICPC.

Statistics	First Edition	Second Edition	Third Edition
Number of Pages	32	32 (+0%)	52 (+63%)
Written Exercises	7	16 (+129%)	11+10*=21 (+31%)
Programming Exercises	109	194 (+78%)	245 (+26%)

The breakdown of the number of programming exercises from each section is shown below:

Section	Title	Appearance	% in Chapter	% in Book
3.2	**Complete Search**	112	45%	7%
3.3	Divide and Conquer	23	9%	1%
3.4	Greedy	45	18%	3%
3.5	**Dynamic Programming**	67	27%	4%

Chapter 4

Graph

Everyone is on average ≈ six steps away from any other person on Earth
— **Stanley Milgram - the Six Degrees of Separation experiment in 1969, [64]**

4.1 Overview and Motivation

Many real-life problems can be classified as graph problems. Some have efficient solutions. Some do not have them yet. In this relatively big chapter with lots of figures, we discuss graph problems that commonly appear in programming contests, the algorithms to solve them, and the *practical* implementations of these algorithms. We cover topics ranging from basic graph traversals, minimum spanning trees, single-source/all-pairs shortest paths, network flows, and discuss graphs with special properties.

In writing this chapter, we assume that the readers are *already* familiar with the graph terminologies listed in Table 4.1. If you encounter any unfamiliar term, please read other reference books like [7, 58] (or browse the Internet) and search for that particular term.

Vertices/Nodes	Edges	Set V; size $|V|$	Set E; size $|E|$	Graph $G(V, E)$
Un/Weighted	Un/Directed	Sparse	Dense	In/Out Degree
Path	Cycle	Isolated	Reachable	Connected
Self-Loop	Multiple Edges	Multigraph	Simple Graph	Sub-Graph
DAG	Tree/Forest	Eulerian	Bipartite	Complete

Table 4.1: List of Important Graph Terminologies

We also assume that the readers have read various ways to represent graph information that have been discussed earlier in Section 2.4.1. That is, we will directly use the terms like: Adjacency Matrix, Adjacency List, Edge List, and implicit graph without redefining them. Please revise Section 2.4.1 if you are not familiar with these graph data structures.

Our research so far on graph problems in recent ACM ICPC (Asia) regional contests reveals that there is at least one (and possibly more) graph problem(s) in an ICPC problem set. However, since the range of graph problems is so big, each graph problem only has a small probability of appearance. So the question is "Which ones do we have to focus on?". In our opinion, there is no clear answer for this question. If you want to do well in ACM ICPC, you have no choice but to study and master all these materials.

For IOI, the syllabus [20] restricts IOI tasks to a subset of material mentioned in this chapter. This is logical as high school students competing in IOI are not expected to be well-versed with too many problem-specific algorithms. To assist the readers aspiring to take part in the IOI, we will mention whether a particular section in this chapter is currently outside the syllabus.

4.2 Graph Traversal

4.2.1 Depth First Search (DFS)

Depth First Search—abbreviated as DFS—is a simple algorithm for traversing a graph. Starting from a distinguished source vertex, DFS will traverse the graph 'depth-first'. Every time DFS hits a branching point (a vertex with more than one neighbors), DFS will choose one of the unvisited neighbor(s) and visit this neighbor vertex. DFS repeats this process and goes deeper until it reaches a vertex where it cannot go any deeper. When this happens, DFS will 'backtrack' and explore another unvisited neighbor(s), if any.

This graph traversal behavior can be implemented easily with the recursive code below. Our DFS implementation uses the help of a *global* vector of integers: `vi dfs_num` to distinguish the state of each vertex. For the simplest DFS implementation, we only use `vi dfs_num` to distinguish between 'unvisited' (we use a constant value `UNVISITED = -1`) and 'visited' (we use another constant value `VISITED = 1`). Initially, all values in `dfs_num` are set to 'unvisited'. We will use `vi dfs_num` for other purposes later. Calling `dfs(u)` starts DFS from a vertex u, marks vertex u as 'visited', and then DFS recursively visits each 'unvisited' neighbor v of u (i.e. edge $u - v$ exists in the graph and `dfs_num[v] == UNVISITED`).

```
typedef pair<int, int> ii; // In this chapter, we will frequently use these
typedef vector<ii> vii; // three data type shortcuts. They may look cryptic
typedef vector<int> vi;   // but they are useful in competitive programming

vi dfs_num;   // global variable, initially all values are set to UNVISITED

void dfs(int u) {      // DFS for normal usage: as graph traversal algorithm
  dfs_num[u] = VISITED;         // important: we mark this vertex as visited
  for (int j = 0; j < (int)AdjList[u].size(); j++) { // default DS: AdjList
    ii v = AdjList[u][j];                   // v is a (neighbor, weight) pair
    if (dfs_num[v.first] == UNVISITED)    // important check to avoid cycle
      dfs(v.first);   // recursively visits unvisited neighbors of vertex u
} }  // for simple graph traversal, we ignore the weight stored at v.second
```

The time complexity of this DFS implementation depends on the graph data structure used. In a graph with V vertices and E edges, DFS runs in $O(V + E)$ and $O(V^2)$ if the graph is stored as Adjacency List and Adjacency Matrix, respectively (see **Exercise 4.2.2.2**).

On the sample graph in Figure 4.1, `dfs(0)`—calling DFS from a starting vertex $u = 0$—will trigger this sequence of visitation: $0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 4$. This sequence is 'depth-first', i.e. DFS goes to the deepest possible vertex from the start vertex before attempting another branch (there is none in this case).

Note that this sequence of visitation depends very much on the way we order the neighbors of a vertex[1], i.e. the sequence $0 \rightarrow 1 \rightarrow 3 \rightarrow 2$ (backtrack to 3) $\rightarrow 4$ is also a possible visitation sequence.

Figure 4.1: Sample Graph

Also notice that one call of `dfs(u)` will only visit all vertices that are *connected* to vertex u. That is why vertices 5, 6, 7, and 8 in Figure 4.1 remain unvisited after calling `dfs(0)`.

---
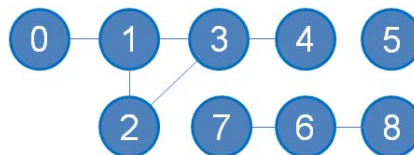
[1]For simplicity, we usually just order the vertices based on their vertex numbers, e.g. in Figure 4.1, vertex 1 has vertex {0, 2, 3} as its neighbor, in that order.

The DFS code shown here is very similar to the recursive backtracking code shown earlier in Section 3.2. If we compare the pseudocode of a typical backtracking code (replicated below) with the DFS code shown above, we can see that the main difference is the flagging of visited vertices (states). Backtracking (automatically) un-flag visited vertices (reset the state to previous state) when the recursion backtracks to allow re-visitation of those vertices (states) from another branch. By not revisiting vertices of a general graph (via `dfs_num` checks), DFS runs in $O(V + E)$, but the time complexity of backtracking is exponential.

```
void backtrack(state) {
  if (hit end state or invalid state)          // we need terminating or
    return;    // pruning condition to avoid cycling and to speed up search
  for each neighbor of this state              // try all permutation
    backtrack(neighbor);
}
```

Sample Application: UVa 11902 - Dominator

Abridged problem description: Vertex X dominates vertex Y if every path from the a start vertex (vertex 0 for this problem) to Y must go through X. If Y is not reachable from the start vertex then Y does not have any dominator. Every vertex reachable from the start vertex dominates itself. For example, in the graph shown in Figure 4.2, vertex 3 dominates vertex 4 since all the paths from vertex 0 to vertex 4 must pass through vertex 3. Vertex 1 does not dominate vertex 3 since there is a path 0-2-3 that does not include vertex 1. Our task: Given a directed graph, determine the dominators of every vertex.

This problem is about reachability tests from a start vertex (vertex 0). Since the input graph for this problem is small ($V < 100$), we can afford to use the following $O(V \times V^2 = V^3)$ algorithm. Run `dfs(0)` on the input graph to record vertices that are reachable from vertex 0. Then to check which vertices are dominated by vertex X, we (temporarily) turn off all the outgoing edges of vertex X and rerun `dfs(0)`. Now, a vertex Y is not dominated by vertex X if `dfs(0)` initially cannot reach vertex Y or `dfs(0)` can reach vertex Y even after all outgoing edges of vertex X are (temporarily) turned off. Vertex Y is dominated by vertex X otherwise. We repeat this process $\forall X \in [0 \ldots V - 1]$.
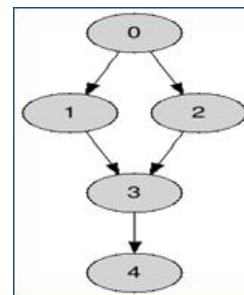
Figure 4.2: UVa 11902

Tips: We do not have to physically delete vertex X from the input graph. We can simply add a statement inside our DFS routine to stop the traversal if it hits vertex X.

4.2.2 Breadth First Search (BFS)

Breadth First Search—abbreviated as BFS—is another graph traversal algorithm. Starting from a distinguished source vertex, BFS will traverse the graph 'breadth-first'. That is, BFS will visit vertices that are direct neighbors of the source vertex (first layer), neighbors of direct neighbors (second layer), and so on, layer by layer.

BFS starts with the insertion of the source vertex s into a queue, then processes the queue as follows: Take out the front most vertex u from the queue, enqueue all unvisited neighbors of u (usually, the neighbors are ordered based on their vertex numbers), and mark them as visited. With the help of the queue, BFS will visit vertex s and all vertices in the connected component that contains s layer by layer. BFS algorithm also runs in $O(V + E)$ and $O(V^2)$

on a graph represented using an Adjacency List and Adjacency Matrix, respectively (again, see **Exercise 4.2.2.2**).

Implementing BFS is easy if we utilize C++ STL or Java API. We use `queue` to order the sequence of visitation and `vector<int>` (or `vi`) to record if a vertex has been visited or not—which at the same time also record the distance (layer number) of each vertex from the source vertex. This distance computation feature is used later to solve a special case of Single-Source Shortest Paths problem (see Section 4.4 and 8.2.3).

```
// inside int main()---no recursion
  vi d(V, INF); d[s] = 0;                    // distance from source s to s is 0
  queue<int> q; q.push(s);                             // start from source

  while (!q.empty()) {
    int u = q.front(); q.pop();                    // queue: layer by layer!
    for (int j = 0; j < (int)AdjList[u].size(); j++) {
      ii v = AdjList[u][j];                        // for each neighbor of u
      if (d[v.first] == INF) {      // if v.first is unvisited + reachable
        d[v.first] = d[u] + 1;          // make d[v.first] != INF to flag it
        q.push(v.first);          // enqueue v.first for the next iteration
  } } }
```
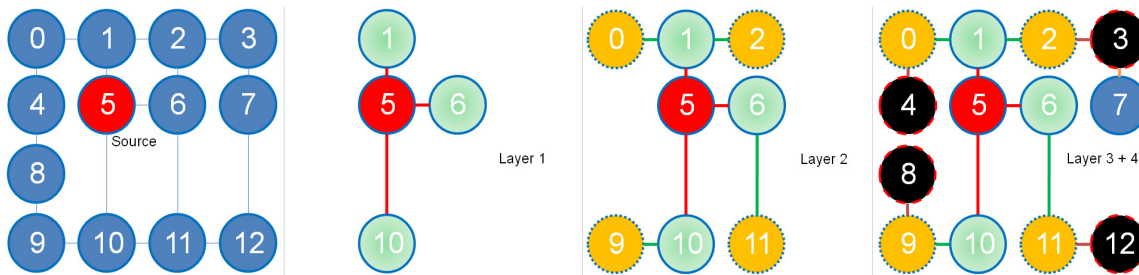

Figure 4.3: Example Animation of BFS

If we run BFS from vertex 5 (i.e. the source vertex $s = 5$) on the connected undirected graph shown in Figure 4.3, we will visit the vertices in the following order:

```
Layer 0:, visit 5
Layer 1:, visit 1, visit 6, visit 10
Layer 2:, visit 0, visit 2, visit 11, visit 9
Layer 3:, visit 4, visit 3, visit 12, visit 8
Layer 4:, visit 7
```

Exercise 4.2.2.1: To show that either DFS or BFS can be used to visit all vertices that are reachable from a source vertex, solve UVa 11902 - Dominator using BFS instead!

Exercise 4.2.2.2: Why do DFS and BFS run in $O(V+E)$ if the graph is stored as Adjacency List and become slower (run in $O(V^2)$) if the graph is stored as Adjacency Matrix? Follow up question: What is the time complexity of DFS and BFS if the graph is stored as Edge List instead? What should we do if the input graph is given as an Edge List and we want to traverse the graph efficiently?

4.2.3 Finding Connected Components (Undirected Graph)

DFS and BFS are not only useful for traversing a graph. They can be used to solve many other graph problems. The first few problems below can be solved with *either* DFS or BFS although some of the last few problems are more suitable for DFS only.

The fact that one single call of `dfs(u)` (or `bfs(u)`) will only visit vertices that are actually connected to u can be utilized to find (and to count the number of) connected components in an *undirected* graph (see further below in Section 4.2.9 for a similar problem on directed graph). We can simply use the following code to restart DFS (or BFS) from one of the remaining unvisited vertices to find the next connected component. This process is repeated until all vertices have been visited and has an overall time complexity of $O(V + E)$.

```
// inside int main()---this is the DFS solution
  numCC = 0;
  dfs_num.assign(V, UNVISITED);    // sets all vertices' state to UNVISITED
  for (int i = 0; i < V; i++)               // for each vertex i in [0..V-1]
    if (dfs_num[i] == UNVISITED)           // if vertex i is not visited yet
      printf("CC %d:", ++numCC), dfs(i), printf("\n");     // 3 lines here!

// For the sample graph in Figure 4.1, the output is like this:
// CC 1: 0 1 2 3 4
// CC 2: 5
// CC 3: 6 7 8
```

Exercise 4.2.3.1: UVa 459 - Graph Connectivity is basically this problem of finding connected components of an undirected graph. Solve it using the DFS solution shown above! However, we can also use Union-Find Disjoint Sets data structure (see Section 2.4.2) or BFS (see Section 4.2.2) to solve this graph problem. How?

4.2.4 Flood Fill - Labeling/Coloring the Connected Components

DFS (or BFS) can be used for other purposes than just finding (and counting the number of) connected components. Here, we show how a *simple tweak* of the $O(V + E)$ `dfs(u)` (we can also use `bfs(u)`) can be used to *label* (also known in CS terminology as 'to color') and count the size of each component. This variant is more famously known as 'flood fill' and usually performed on *implicit* graphs (usually 2D grids).

```
int dr[] = {1,1,0,-1,-1,-1, 0, 1};  // trick to explore an implicit 2D grid
int dc[] = {0,1,1, 1, 0,-1,-1,-1};         // S,SE,E,NE,N,NW,W,SW neighbors

int floodfill(int r, int c, char c1, char c2) {   // returns the size of CC
  if (r < 0 || r >= R || c < 0 || c >= C) return 0;        // outside grid
  if (grid[r][c] != c1) return 0;                 // does not have color c1
  int ans = 1;   // adds 1 to ans because vertex (r, c) has c1 as its color
  grid[r][c] = c2;    // now recolors vertex (r, c) to c2 to avoid cycling!
  for (int d = 0; d < 8; d++)
    ans += floodfill(r + dr[d], c + dc[d], c1, c2);
  return ans;                        // the code is neat due to dr[] and dc[]
}
```

Sample Application: UVa 469 - Wetlands of Florida

Let's see an example below (UVa 469 - Wetlands of Florida). The implicit graph is a 2D grid where the vertices are the cells in the grid and the edges are the connections between a cell and its S/SE/E/NE/N/NW/W/SW cells. 'W' denotes a wet cell and 'L' denotes a land cell. Wet area is defined as *connected cells* labeled with 'W'. We can label (and simultaneously count the size of) a wet area by using floodfill. The example below shows an execution of floodfill from row 2, column 1 (0-based indexing), replacing 'W' to '.'.

   We want to make a remark that there are a good number of floodfill problems in UVa online judge [47] with a high profile example: UVa 1103 - Ancient Messages (ICPC World Finals problem in 2011). It may be beneficial for the readers to attempt floodfill problems listed in programming exercises of this section to master this technique!

```
// inside int main()
  // read the grid as a global 2D array + read (row, col) query coordinates
  printf("%d\n", floodfill(row, col, 'W', '.'));  // count size of wet area
                                      // the returned answer is 12
// LLLLLLLLL          LLLLLLLLL
// LLWWLLWLL          LL..LLWLL  //          The size of connected component
// LWWLLLLLL (R2,C1) L..LLLLLL  //                    (the connected 'W's)
// LWWWLWWLL          L...L..LL  // with one 'W' at (row 2, column 1) is 12
// LLLWWWLLL ======> LLL...LLL
// LLLLLLLLL          LLLLLLLLL  //     Notice that all these connected 'W's
// LLLWWLLWL          LLLWWLLWL  //   are replaced with '.'s after floodfill
// LLWLWLLLL          LLWLWLLLL
// LLLLLLLLL          LLLLLLLLL
```

4.2.5   Topological Sort (Directed Acyclic Graph)

Topological sort (or topological ordering) of a Directed Acyclic Graph (DAG) is a linear ordering of the vertices in the DAG so that vertex u comes before vertex v if edge $(u \to v)$ exists in the DAG. Every DAG has at least one *and possibly more* topological sort(s).

   One application of topological sorting is to find a possible sequence of modules that a University student has to take to fulfill graduation requirement. Each module has certain pre-requisites to be met. These pre-requisites are never cyclic, so they can be modeled as a DAG. Topological sorting this module pre-requisites DAG gives the student a linear list of modules to be taken one after another without violating the pre-requisites constraints.

   There are several algorithms for topological sort. The simplest way is to slightly modify the DFS implementation we presented earlier in Section 4.2.1.

```
vi ts;              // global vector to store the toposort in reverse order

void dfs2(int u) {  // different function name compared to the original dfs
  dfs_num[u] = VISITED;
  for (int j = 0; j < (int)AdjList[u].size(); j++) {
    ii v = AdjList[u][j];
    if (dfs_num[v.first] == UNVISITED)
      dfs2(v.first);
  }
  ts.push_back(u); }                      // that's it, this is the only change
```

```
// inside int main()
  ts.clear();
  memset(dfs_num, UNVISITED, sizeof dfs_num);
  for (int i = 0; i < V; i++)        // this part is the same as finding CCs
    if (dfs_num[i] == UNVISITED)
      dfs2(i);
                // alternative, call: reverse(ts.begin(), ts.end()); first
  for (int i = (int)ts.size() - 1; i >= 0; i--)          // read backwards
    printf(" %d", ts[i]);
  printf("\n");

// For the sample graph in Figure 4.4, the output is like this:
// 7 6 0 1 2 5 3 4    (remember that there can be >= 1 valid toposort)
```

In `dfs2(u)`, we append u to the back of a list (vector) of explored vertices only after visiting all the subtrees below u in the DFS spanning tree[2]. We append u to the *back* of this vector because C++ STL `vector` (Java `Vector`) only supports *efficient* $O(1)$ *insertion* from the back. The list will be in reversed order, but we can work around this issue by reversing the print order in the output phase. This simple algorithm for finding (a valid) topological sort is due to Robert Endre Tarjan. It runs in $O(V + E)$ as with DFS as it does the same work as the original DFS plus one constant operation.
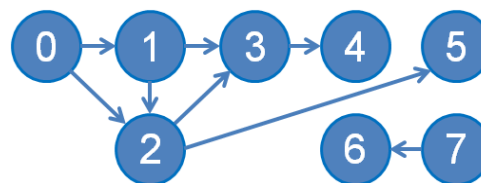
Figure 4.4: An Example of DAG

To complete the discussion about topological sort, we show another algorithm for finding topological sort: Kahn's algorithm [36]. It looks like a 'modified BFS'. Some problems, e.g. UVa 11060 - Beverages, requires this Kahn's algorithm to produce the required topological sort instead of the DFS-based algorithm shown earlier.

```
enqueue vertices with zero incoming degree into a (priority) queue Q;
while (Q is not empty) {
  vertex u = Q.dequeue(); put vertex u into a topological sort list;
  remove this vertex u and all outgoing edges from this vertex;
  if such removal causes vertex v to have zero incoming degree
    Q.enqueue(v); }
```

Exercise 4.2.5.1: Why appending vertex `u` at the back of `vi ts`, i.e. `ts.push_back(u)` in the standard DFS code is enough to help us find the topological sort of a DAG?

Exercise 4.2.5.2: Can you identify another data structure that supports efficient $O(1)$ insertion *from front* so that we do not have to reverse the content of `vi ts`?

Exercise 4.2.5.3: What happen if we run topological sort code above on a non DAG?

Exercise 4.2.5.4: The topological sort code shown above can only generate *one* valid topological ordering of the vertices of a DAG. What should we do if we want to output *all* valid topological orderings of the vertices of a DAG?

---

[2]DFS spanning tree is discussed in more details in Section 4.2.7.

4.2.6   Bipartite Graph Check

Bipartite graph has important applications that we will see later in Section 4.7.4. In this subsection, we just want to check if a graph is bipartite (or 2/bi-colorable) to solve problems like UVa 10004 - Bicoloring. We can use either BFS or DFS for this check, but we feel that BFS is more natural. The modified BFS code below starts by coloring the source vertex (first layer) with value 0, color the direct neighbors of the source vertex (second layer) with value 1, color the neighbors of direct neighbors (third layer) with value 0 again, and so on, alternating between value 0 and value 1 as the only two valid colors. If we encounter any violation(s) along the way—an edge with two endpoints having the same color, then we can conclude that the given input graph is not a bipartite graph.

```
// inside int main()
  queue<int> q; q.push(s);
  vi color(V, INF); color[s] = 0;
  bool isBipartite = true;  // addition of one boolean flag, initially true
  while (!q.empty() & isBipartite) { // similar to the original BFS routine
    int u = q.front(); q.pop();
    for (int j = 0; j < (int)AdjList[u].size(); j++) {
      ii v = AdjList[u][j];
      if (color[v.first] == INF) {   // but, instead of recording distance,
        color[v.first] = 1 - color[u];  // we just record two colors {0, 1}
        q.push(v.first); }
      else if (color[v.first] == color[u]) {  // u & v.first has same color
        isBipartite = false; break; } } }   // we have a coloring conflict
```

Exercise 4.2.6.1*: Implement bipartite check using DFS instead!

Exercise 4.2.6.2*: A *simple* graph with V vertices is found out to be a bipartite graph. What is the maximum possible number of edges that this graph has?

Exercise 4.2.6.3: Prove (or disprove) this statement: "Bipartite graph has no odd cycle"!

4.2.7   Graph Edges Property Check via DFS Spanning Tree

Running DFS on a connected graph generates a DFS *spanning tree*[3] (or *spanning forest*[4] if the graph is disconnected). With the help of one more vertex state: `EXPLORED = 2` (visited *but not yet completed*) on top of `VISITED` (visited *and completed*), we can use this DFS spanning tree (or forest) to classify graph edges into three types:

1. Tree edge: The edge traversed by DFS, i.e. an edge from a vertex currently with state: `EXPLORED` to a vertex with state: `UNVISITED`.

2. Back edge: Edge that is part of a cycle, i.e. an edge from a vertex currently with state: `EXPLORED` to a vertex with state: `EXPLORED` too. This is an important application of this algorithm. Note that we usually do not count bi-directional edges as having a 'cycle' (We need to remember `dfs_parent` to distinguish this, see the code below).

3. Forward/Cross edges from vertex with state: `EXPLORED` to vertex with state: `VISITED`. These two type of edges are not typically tested in programming contest problems.
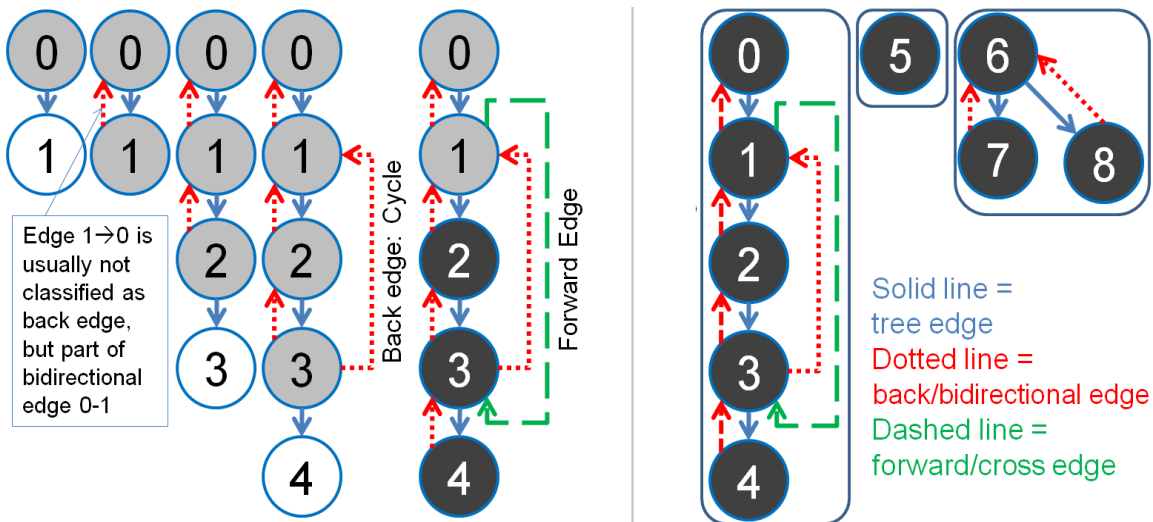
Figure 4.5: Animation of DFS when Run on the Sample Graph in Figure 4.1

Figure 4.5 shows an animation (from left to right) of calling `dfs(0)` (shown in more details), then `dfs(5)`, and finally `dfs(6)` on the sample graph in Figure 4.1. We can see that $1 \to 2 \to 3 \to 1$ is a (true) cycle and we classify edge $(3 \to 1)$ as a back edge, whereas $0 \to 1 \to 0$ is not a cycle but it is just a bi-directional edge (0-1). The code for this DFS variant is shown below.

```
void graphCheck(int u) {               // DFS for checking graph edge properties
  dfs_num[u] = EXPLORED;               // color u as EXPLORED instead of VISITED
  for (int j = 0; j < (int)AdjList[u].size(); j++) {
    ii v = AdjList[u][j];
    if (dfs_num[v.first] == UNVISITED) {  // Tree Edge, EXPLORED->UNVISITED
      dfs_parent[v.first] = u;            // parent of this children is me
      graphCheck(v.first);
    }
    else if (dfs_num[v.first] == EXPLORED) {        // EXPLORED->EXPLORED
      if (v.first == dfs_parent[u])     // to differentiate these two cases
        printf(" Two ways (%d, %d)-(%d, %d)\n", u, v.first, v.first, u);
      else   // the most frequent application: check if the graph is cyclic
        printf(" Back Edge (%d, %d) (Cycle)\n", u, v.first);
    }
    else if (dfs_num[v.first] == VISITED)              // EXPLORED->VISITED
      printf(" Forward/Cross Edge (%d, %d)\n", u, v.first);
  }
  dfs_num[u] = VISITED;        // after recursion, color u as VISITED (DONE)
}

// inside int main()
  dfs_num.assign(V, UNVISITED);
  dfs_parent.assign(V, 0);                                      // new vector
```

---

[3]A spanning tree of a connected graph G is a tree that spans (covers) all vertices of G but only using a subset of the edges of G.

[4]A disconnected graph G has several connected components. Each component has its own spanning subtree(s). All spanning subtrees of G, one from each component, form what we call a spanning forest.

```
  for (int i = 0; i < V; i++)
    if (dfs_num[i] == UNVISITED)
      printf("Component %d:\n", ++numComp), graphCheck(i); // 2 lines in 1!

// For the sample graph in Figure 4.1, the output is like this:
// Component 1:
//  Two ways (1, 0) - (0, 1)
//  Two ways (2, 1) - (1, 2)
//  Back Edge (3, 1) (Cycle)
//  Two ways (3, 2) - (2, 3)
//  Two ways (4, 3) - (3, 4)
//  Forward/Cross Edge (1, 3)
// Component 2:
// Component 3:
//  Two ways (7, 6) - (6, 7)
//  Two ways (8, 6) - (6, 8)
```

Exercise 4.2.7.1: Perform graph edges property check on the graph in Figure 4.9. Assume that you start DFS from vertex 0. How many back edges that you can find this time?

4.2.8   Finding Articulation Points and Bridges (Undirected Graph)

Motivating problem: Given a road map (undirected graph) with sabotage costs associated to all intersections (vertices) and roads (edges), sabotage either a single intersection or a single road such that the road network breaks down (disconnected) and do so in the least cost way. This is a problem of finding the least cost Articulation Point (intersection) or the least cost Bridge (road) in an undirected graph (road map).

An 'Articulation Point' is defined as *a vertex* in a graph G whose removal (all edges incident to this vertex are also removed) disconnects G. A graph without any articulation point is called 'Biconnected'. Similarly, a 'Bridge' is defined as *an edge* in a graph G whose removal disconnects G. These two problems are usually defined for undirected graphs (they are more challenging for directed graphs and require another algorithm to solve, see [35]).

A naïve algorithm to find articulation points is as follows (can be tweaked to find bridges):

1. Run $O(V + E)$ DFS (or BFS) to count number of connected components (CCs) of the original graph. Usually, the input is a connected graph, so this check will usually gives us one connected component.

2. For each vertex $v \in V$ // $O(V)$

   (a) Cut (remove) vertex v and its incident edges
   (b) Run $O(V + E)$ DFS (or BFS) and see if the number of CCs increases
   (c) If yes, v is an articulation point/cut vertex; Restore v and its incident edges

This naïve algorithm calls DFS (or BFS) $O(V)$ times, thus it runs in $O(V \times (V + E)) = O(V^2 + VE)$. But this is *not* the best algorithm as we can actually just run the $O(V + E)$ DFS *once* to identify all the articulation points and bridges.

This DFS variant, due to John Edward Hopcroft and Robert Endre Tarjan (see [63] and problem 22.2 in [7]), is just another extension from the previous DFS code shown earlier.

We now maintain two numbers: `dfs_num(u)` and `dfs_low(u)`. Here, `dfs_num(u)` now stores the iteration counter when the vertex u is visited *for the first time* (not just for distinguishing `UNVISITED` versus `EXPLORED/VISITED`). The other number `dfs_low(u)` stores the lowest `dfs_num` reachable from the current DFS spanning subtree of u. At the beginning, `dfs_low(u) = dfs_num(u)` when vertex u is visited for the first time. Then, `dfs_low(u)` can only be made smaller if there is a cycle (a back edge exists). Note that we do not update `dfs_low(u)` with a back edge (u, v) if v is a direct parent of u.
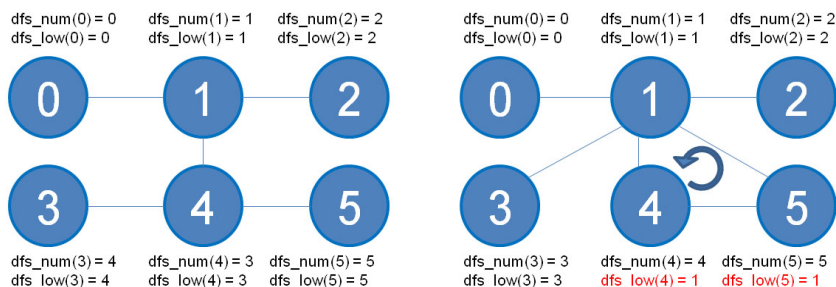
Figure 4.6: Introducing two More DFS Attributes: `dfs_num` and `dfs_low`

See Figure 4.6 for clarity. In both graphs, we run the DFS variant from vertex 0. Suppose for the graph in Figure 4.6—left side, the sequence of visitation is 0 (at iteration 0) → 1 (1) → 2 (2) (backtrack to 1) → 4 (3) → 3 (4) (backtrack to 4) → 5 (5). See that these iteration counters are shown correctly in `dfs_num`. As there is no back edge in this graph, all `dfs_low = dfs_num`.

Suppose for the graph in Figure 4.6—right side, the sequence of visitation is 0 (at iteration 0) → 1 (1) → 2 (2) (backtrack to 1) → 3 (3) (backtrack to 1) → 4 (4) → 5 (5). At this point in the DFS spanning tree, there is an important back edge that forms a cycle, i.e. edge 5-1 that is part of cycle 1-4-5-1. This causes vertices 1, 4, and 5 to be able to reach vertex 1 (with `dfs_num` 1). Thus `dfs_low` of {1, 4, 5} are all 1.

When we are in a vertex u with v as its neighbor and `dfs_low(v) ≥ dfs_num(u)`, then u is an articulation vertex. This is because the fact that `dfs_low(v)` is *not smaller* than `dfs_num(u)` implies that there is *no back edge* from vertex v that can reach another vertex w with a lower `dfs_num(w)` than `dfs_num(u)`. A vertex w with lower `dfs_num(w)` than vertex u with `dfs_num(u)` implies that w is the ancestor of u in the DFS spanning tree. This means that to reach the ancestor(s) of u from v, one *must* pass through vertex u. Therefore, removing vertex u will disconnect the graph.

However, there is one **special case**: The root of the DFS spanning tree (the vertex chosen as the start of DFS call) is an articulation point only if it has more than one children in the DFS spanning tree (a trivial case that is not detected by this algorithm).
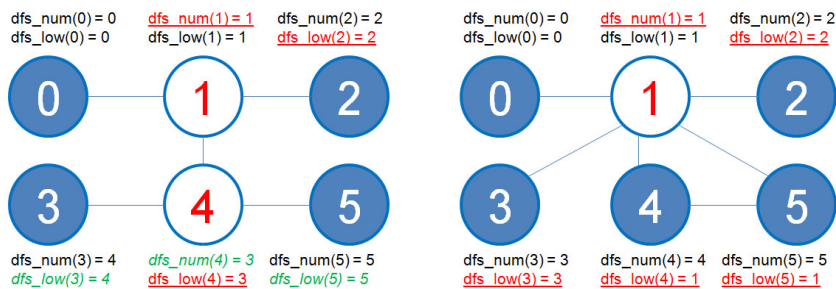
Figure 4.7: Finding Articulation Points with `dfs_num` and `dfs_low`

See Figure 4.7 for more details. On the graph in Figure 4.7—left side, vertices 1 and 4 are articulation points, because for example in edge 1-2, we see that `dfs_low(2) ≥ dfs_num(1)`

and in edge 4-5, we also see that `dfs_low(5)` $\geq$ `dfs_num(4)`. On the graph in Figure 4.7—right side, only vertex 1 is the articulation point, because for example in edge 1-5, `dfs_low(5)` $\geq$ `dfs_num(1)`.
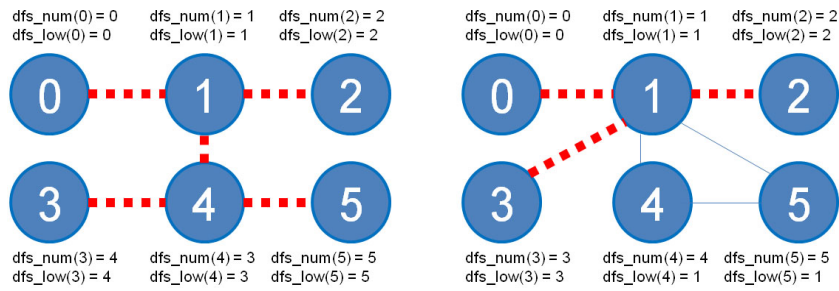
Figure 4.8: Finding Bridges, also with `dfs_num` and `dfs_low`

The process to find bridges is similar. When `dfs_low(v)` > `dfs_num(u)`, then edge u-v is a bridge (notice that we remove the equality test '=' for finding bridges). In Figure 4.8, almost all edges are bridges for the left and right graph. Only edges 1-4, 4-5, and 5-1 are not bridges on the right graph (they actually form a cycle). This is because—for example—for edge 4-5, we have `dfs_low(5)` $\leq$ `dfs_num(4)`, i.e. even if this edge 4-5 is removed, we know for sure that vertex 5 can still reach vertex 1 via *another path* that bypass vertex 4 as `dfs_low(5)` = 1 (that other path is actually edge 5-1). The code is shown below:

```
void articulationPointAndBridge(int u) {
  dfs_low[u] = dfs_num[u] = dfsNumberCounter++; // dfs_low[u] <= dfs_num[u]
  for (int j = 0; j < (int)AdjList[u].size(); j++) {
    ii v = AdjList[u][j];
    if (dfs_num[v.first] == UNVISITED) {                    // a tree edge
      dfs_parent[v.first] = u;
      if (u == dfsRoot) rootChildren++;     // special case if u is a root

      articulationPointAndBridge(v.first);

      if (dfs_low[v.first] >= dfs_num[u])         // for articulation point
        articulation_vertex[u] = true;      // store this information first
      if (dfs_low[v.first] > dfs_num[u])                      // for bridge
        printf(" Edge (%d, %d) is a bridge\n", u, v.first);
      dfs_low[u] = min(dfs_low[u], dfs_low[v.first]);  // update dfs_low[u]
    }
    else if (v.first != dfs_parent[u])  // a back edge and not direct cycle
      dfs_low[u] = min(dfs_low[u], dfs_num[v.first]);  // update dfs_low[u]
} }

// inside int main()
  dfsNumberCounter = 0; dfs_num.assign(V, UNVISITED); dfs_low.assign(V, 0);
  dfs_parent.assign(V, 0); articulation_vertex.assign(V, 0);
  printf("Bridges:\n");
  for (int i = 0; i < V; i++)
    if (dfs_num[i] == UNVISITED) {
      dfsRoot = i; rootChildren = 0; articulationPointAndBridge(i);
      articulation_vertex[dfsRoot] = (rootChildren > 1); }  // special case
```

```
printf("Articulation Points:\n");
for (int i = 0; i < V; i++)
  if (articulation_vertex[i])
    printf(" Vertex %d\n", i);
```

Exercise 4.2.8.1: Examine the graph in Figure 4.1 without running the algorithm above. Which vertices are articulation points and which edges are bridges? Now run the algorithm and verify if the computed `dfs_num` and `dfs_low` of each vertex of Figure 4.1 graph can be used to identify the same articulation points and bridges found manually!

4.2.9 Finding Strongly Connected Components (Directed Graph)

Yet another application of DFS is to find *strongly* connected components in a *directed* graph, e.g. UVa 11838 - Come and Go. This is a different problem to finding connected components in an undirected graph. In Figure 4.9, we have a similar graph to the graph in Figure 4.1, but now the edges are directed. Although the graph in Figure 4.9 looks like it has one 'connected' component, it is actually not a 'strongly connected' component. In directed graphs, we are more interested with the notion of 'Strongly Connected Component (SCC)'. An SCC is defined as such: If we pick any pair of vertices u and v in the SCC, we can find a path from u to v and vice versa. There are actually three SCCs in Figure 4.9, as highlighted with the three boxes: {0}, {1, 3, 2}, and {4, 5, 7, 6}. Note: If these SCCs are contracted (replaced by larger vertices), they form a DAG (also see Section 8.4.3).

There are at least two known algorithms to find SCCs: Kosaraju's—explained in [7] and Tarjan's algorithm [63]. In this section, we adopt Tarjan's version, as it extends naturally from our previous discussion of finding Articulation Points and Bridges—also due to Tarjan. We will discuss Kosaraju's algorithm later in Section 9.17.

The basic idea of the algorithm is that SCCs form subtrees in the DFS spanning tree (compare the original directed graph and the DFS spanning tree in Figure 4.9). On top of computing `dfs_num(u)` and `dfs_low(u)` for each vertex, we also append vertex u to the back of a stack S (here the stack is implemented with a vector) and keep track of the vertices that are currently explored via `vi visited`. The condition to update `dfs_low(u)` is slightly different from the previous DFS algorithm for finding articulation points and bridges. Here, only vertices that currently have `visited` flag turned on (part of the current SCC) that can update `dfs_low(u)`. Now, if we have vertex u in this DFS spanning tree with `dfs_low(u) = dfs_num(u)`, we can conclude that u is the root (start) of an SCC (observe vertex 0, 1, and 4) in Figure 4.9) and the members of those SCCs are identified by popping the current content of stack S until we reach vertex u (the root) of SCC again.

In Figure 4.9, the content of S is $\{0, 1, 3, 2, \underline{4, 5, 7, 6}\}$ when vertex 4 is identified as the root of an SCC (`dfs_low(4) = dfs_num(4) = 4`), so we pop elements in S one by one until we reach vertex 4 and we have this SCC: {6, 7, 5, 4}. Next, the content of S is $\{0, \underline{1, 3, 2}\}$ when vertex 1 is identified as another root of another SCC (`dfs_low(1) = dfs_num(1) = 1`), so we pop elements in S one by one until we reach vertex 1 and we have SCC: {2, 3, 1}. Finally, we have the last SCC with one member only: {0}.

The code given below explores the directed graph and reports its SCCs. This code is basically a tweak of the standard DFS code. The recursive part is similar to standard DFS and the SCC reporting part will run in amortized $O(V)$ times, as each vertex will only belong to one SCC and thus reported only once. In overall, this algorithm still runs in $O(V + E)$.
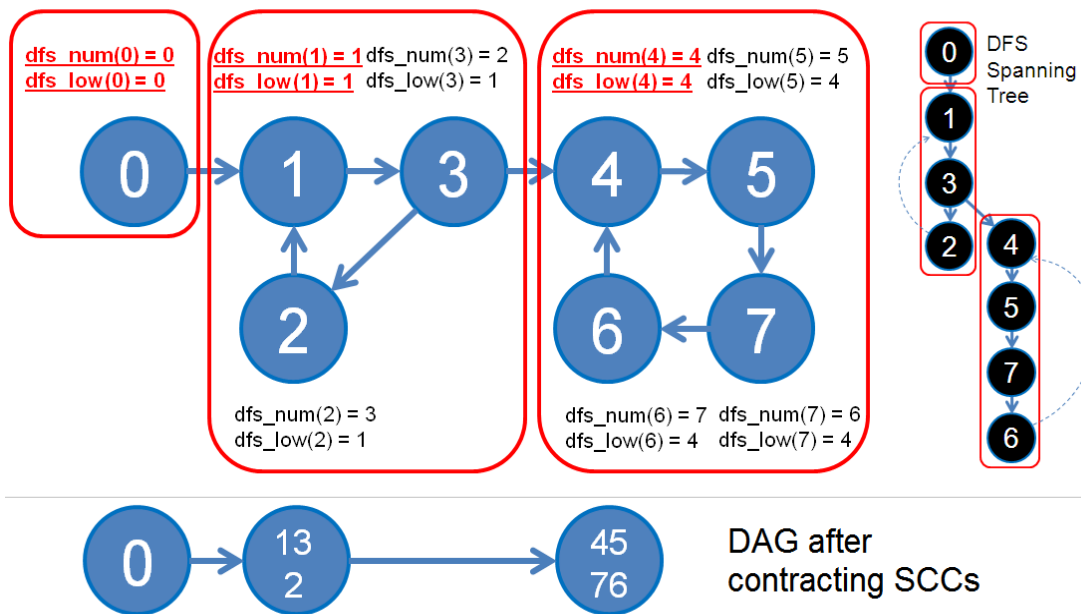
Figure 4.9: An Example of a Directed Graph and its SCCs

```
vi dfs_num, dfs_low, S, visited;                        // global variables

void tarjanSCC(int u) {
  dfs_low[u] = dfs_num[u] = dfsNumberCounter++; // dfs_low[u] <= dfs_num[u]
  S.push_back(u);        // stores u in a vector based on order of visitation
  visited[u] = 1;
  for (int j = 0; j < (int)AdjList[u].size(); j++) {
    ii v = AdjList[u][j];
    if (dfs_num[v.first] == UNVISITED)
      tarjanSCC(v.first);
    if (visited[v.first])                              // condition for update
      dfs_low[u] = min(dfs_low[u], dfs_low[v.first]); }

  if (dfs_low[u] == dfs_num[u]) {    // if this is a root (start) of an SCC
    printf("SCC %d:", ++numSCC);        // this part is done after recursion
    while (1) {
      int v = S.back(); S.pop_back(); visited[v] = 0;
      printf(" %d", v);
      if (u == v) break; }
    printf("\n");
} }

// inside int main()
  dfs_num.assign(V, UNVISITED); dfs_low.assign(V, 0); visited.assign(V, 0);
  dfsNumberCounter = numSCC = 0;
  for (int i = 0; i < V; i++)
    if (dfs_num[i] == UNVISITED)
      tarjanSCC(i);
```

Source code: ch4_01_dfs.cpp/java; ch4_02_UVa469.cpp/java

Exercise 4.2.9.1: Prove (or disprove) this statement: "If two vertices are in the same SCC, then there is no path between them that ever leaves the SCC"!

Exercise 4.2.9.2*: Write a code that takes in a Directed Graph and then convert it into a Directed Acyclic Graph (DAG) by contracting the SCCs (e.g Figure 4.9, top to bottom)! See Section 8.4.3 for a sample application.

Remarks About Graph Traversal in Programming Contests

It is remarkable that the simple DFS and BFS traversal algorithms have so many interesting variants that can be used to solve various graph problems on top of their basic form for traversing a graph. In ICPC, any of these variants can appear. In IOI, creative tasks involving graph traversal can appear.

Using DFS (or BFS) to find connected components in an undirected graph is rarely asked per se although its variant: flood fill, is one of the most frequent problem type *in the past*. However, we feel that the number of (new) flood fill problems is getting smaller.

Topological sort is rarely used per se, but it is a useful pre-processing step for 'DP on (implicit) DAG', see Section 4.7.1. The simplest version of topological sort code is very easy to memorize as it is just a simple DFS variant. The alternative Kahn's algorithm (the 'modified BFS' that only enqueue vertices with 0-incoming degrees) is also equally simple.

Efficient $O(V + E)$ solutions for bipartite graph check, graph edges property check, and finding articulation points/bridges are good to know but as seen in the UVa online judge (and recent ICPC regionals in Asia), not many problems use them now.

The knowledge of Tarjan's SCC algorithm may come in handy to solve modern problems where one of its sub-problem involves directed graphs that 'requires transformation' to DAG by contracting cycles—see Section 8.4.3. The library code shown in this book may be something that you should bring into a programming contest that allows hard copy printed library code like ICPC. However in IOI, the topic of Strongly Connected Component is currently excluded from the IOI 2009 syllabus [20].

Although many of the graph problems discussed in this section can be solved by either DFS or BFS. Personally, we feel that many of them are easier to be solved using the recursive and more memory friendly DFS. We do not normally use BFS for pure graph traversal problems but we will use it to solve the Single-Source Shortest Paths problems on unweighted graph (see Section 4.4). Table 4.2 shows important comparison between these two popular graph traversal algorithms.

	$O(V + E)$ **DFS**	$O(V + E)$ **BFS**
Pros	*Usually* use less memory	Can solve SSSP
	Can find articulation points, bridges, SCC	(on unweighted graphs)
Cons	Cannot solve SSSP	*Usually* use more memory
	on unweighted graphs	(bad for large graph)
Code	Slightly easier to code	Just a bit longer to code

Table 4.2: Graph Traversal Algorithm Decision Table

We have provided the animation of DFS/BFS algorithm and (some of) their variants in the URL below. Use it to further strengthen your understanding of these algorithms.

Visualization: `www.comp.nus.edu.sg/~stevenha/visualization/dfsbfs.html`

Programming Exercises related to Graph Traversal:

- Just Graph Traversal

    1. UVa 00118 - Mutant Flatworld Explorers (traversal on *implicit* graph)
    2. UVa 00168 - Theseus and the ... (Adjacency Matrix, parsing, traversal)
    3. UVa 00280 - Vertex (graph, reachability test by traversing the graph)
    4. *UVa 00318 - Domino Effect* (traversal, be careful of corner cases)
    5. UVa 00614 - Mapping the Route (traversal on *implicit* graph)
    6. UVa 00824 - Coast Tracker (traversal on *implicit* graph)
    7. UVa 10113 - Exchange Rates (just graph traversal, but uses fraction and gcd, see the relevant sections in Chapter 5)
    8. UVa 10116 - Robot Motion (traversal on *implicit* graph)
    9. UVa 10377 - Maze Traversal (traversal on *implicit* graph)
    10. UVa 10687 - Monitoring the Amazon (build graph, geometry, reachability)
    11. **UVa 11831 - Sticker Collector ... *** (*implicit* graph; input order is 'NSEW'!)
    12. UVa 11902 - Dominator (disable vertex one by one, check if the reachability from vertex 0 changes)
    13. **UVa 11906 - Knight in a War Grid *** (DFS/BFS for reachability, several tricky cases; be careful when M = 0 ∥ N = 0 ∥ M = N)
    14. *UVa 12376 - As Long as I Learn, I Live* (simulated greedy traversal on DAG)
    15. ***UVa 12442 - Forwarding Emails **** (modified DFS, special graph)
    16. *UVa 12582 - Wedding of Sultan* (given graph DFS traversal, count the degree of each vertex)
    17. IOI 2011 - Tropical Garden (graph traversal; DFS; involving cycle)

- Flood Fill/Finding Connected Components

    1. UVa 00260 - Il Gioco dell'X (6 neighbors per cell!)
    2. UVa 00352 - The Seasonal War (count # of connected components (CC))
    3. UVa 00459 - Graph Connectivity (also solvable with 'union find')
    4. UVa 00469 - Wetlands of Florida (count size of a CC; discussed in this section)
    5. UVa 00572 - Oil Deposits (count number of CCs, similar to UVa 352)
    6. UVa 00657 - The Die is Cast (there are three 'colors' here)
    7. *UVa 00722 - Lakes* (count the size of CCs)
    8. *UVa 00758 - The Same Game* (floodfill++)
    9. UVa 00776 - Monkeys in a Regular ... (label CCs with indices, format output)
    10. UVa 00782 - Countour Painting (replace ' ' with '#' in the grid)
    11. UVa 00784 - Maze Exploration (very similar with UVa 782)
    12. UVa 00785 - Grid Colouring (also very similar with UVa 782)
    13. UVa 00852 - Deciding victory in Go (interesting board game 'Go')
    14. UVa 00871 - Counting Cells in a Blob (find the size of the largest CC)
    15. ***UVa 01103 - Ancient Messages **** (LA 5130, World Finals Orlando11; major hint: each hieroglyph has unique number of white connected component; then it is an implementation exercise to parse the input and run flood fill to determine the number of white CC inside each black hieroglyph)
    16. UVa 10336 - Rank the Languages (count and rank CCs with similar color)

17. UVa 10707 - 2D - Nim (check graph isomorphism; a tedious problem; involving connected components)
18. UVa 10946 - You want what filled? (find CCs and rank them by their size)
19. **UVa 11094 - Continents *** (tricky flood fill as it involves scrolling)
20. UVa 11110 - Equidivisions (flood fill + satisfy the constraints given)
21. UVa 11244 - Counting Stars (count number of CCs)
22. UVa 11470 - Square Sums (you can do 'flood fill' layer by layer; however, there is other way to solve this problem, e.g. by finding the patterns)
23. UVa 11518 - Dominos 2 (unlike UVa 11504, we treat SCCs as simple CCs)
24. UVa 11561 - Getting Gold (flood fill with extra blocking constraint)
25. UVa 11749 - Poor Trade Advisor (find largest CC with highest average PPA)
26. **UVa 11953 - Battleships *** (interesting twist of flood fill problem)

- Topological Sort

   1. UVa 00124 - Following Orders (use backtracking to generate valid toposorts)
   2. UVa 00200 - Rare Order (toposort)
   3. **UVa 00872 - Ordering *** (similar to UVa 124, use backtracking)
   4. **UVa 10305 - Ordering Tasks *** (run toposort algorithm in this section)
   5. **UVa 11060 - Beverages *** (must use Kahn's algorithm—the 'modified BFS' topological sort)
   6. UVa 11686 - Pick up sticks (toposort + cycle check)
      Also see: DP on (implicit) DAG problems (see Section 4.7.1)

- Bipartite Graph Check

   1. **UVa 10004 - Bicoloring *** (bipartite graph check)
   2. UVa 10505 - Montesco vs Capuleto (bipartite graph, take max(left, right))
   3. **UVa 11080 - Place the Guards *** (bipartite graph check, some tricky cases)
   4. **UVa 11396 - Claw Decomposition *** (it is just a bipartite graph check)

- Finding Articulation Points/Bridges

   1. **UVa 00315 - Network *** (finding articulation points)
   2. UVa 00610 - Street Directions (finding bridges)
   3. **UVa 00796 - Critical Links *** (finding bridges)
   4. UVa 10199 - Tourist Guide (finding articulation points)
   5. ***UVa 10765 - Doves and Bombs **** (finding articulation points)

- Finding Strongly Connected Components

   1. **UVa 00247 - Calling Circles *** (SCC + printing solution)
   2. UVa 01229 - Sub-dictionary (LA 4099, Iran07, identify the SCC of the graph; these vertices and the vertices that have path towards them (e.g. needed to understand these words too) are the answers of the question)
   3. UVa 10731 - Test (SCC + printing solution)
   4. **UVa 11504 - Dominos *** (interesting problem: count |SCCs| without incoming edge from a vertex outside that SCC)
   5. UVa 11709 - Trust Groups (find number of SCC)
   6. UVa 11770 - Lighting Away (similar to UVa 11504)
   7. **UVa 11838 - Come and Go *** (check if graph is strongly connected)

4.3 Minimum Spanning Tree

4.3.1 Overview and Motivation

Motivating problem: Given a connected, undirected, and weighted graph G (see the leftmost graph in Figure 4.10), select a subset of edges $E' \in G$ such that the graph G is (still) connected and the total weight of the selected edges E' is minimal!
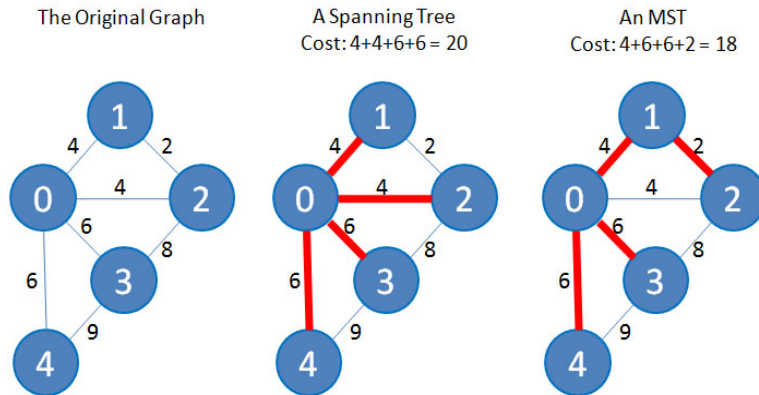
Figure 4.10: Example of an MST Problem

To satisfy the connectivity criteria, we need at least $V-1$ edges that form a *tree* and this tree must spans (covers) all $V \in G$—the *spanning tree*! There can be several valid spanning trees in G, i.e. see Figure 4.10, middle and right sides. The DFS and BFS spanning trees that we have learned in previous Section 4.2 are also possible. Among these possible spanning trees, there are some (at least one) that satisfy the minimal weight criteria.

This problem is called the Minimum Spanning Tree (MST) problem and has many practical applications. For example, we can model a problem of building road network in remote villages as an MST problem. The vertices are the villages. The edges are the potential roads that may be built between those villages. The cost of building a road that connects village i and j is the weight of edge (i, j). The MST of this graph is therefore the minimum cost road network that connects all these villages. In UVa online judge [47], we have some basic MST problems like this, e.g. UVa 908, 1174, 1208, 10034, 11631, etc.

This MST problem can be solved with several well-known algorithms, i.e. Prim's and Kruskal's. Both are Greedy algorithms and explained in many CS textbooks [7, 58, 40, 60, 42, 1, 38, 8]. The MST weight produced by these two algorithms is unique, but there can be more than one spanning tree that have the same MST weight.

4.3.2 Kruskal's Algorithm

Joseph Bernard *Kruskal* Jr.'s algorithm first sorts E edges based on non decreasing weight. This can be easily done by storing the edges in an EdgeList data structure (see Section 2.4.1) and then sort the edges based on non-decreasing weight. Then, Kruskal's algorithm *greedily* tries to add each edge into the MST as long as such addition does not form a cycle. This cycle check can be done easily using the lightweight Union-Find Disjoint Sets discussed in Section 2.4.2. The code is short (because we have separated the Union-Find Disjoint Sets implementation code in a separate class). The overall runtime of this algorithm is O(sorting + trying to add each edge × cost of Union-Find operations) = $O(E \log E + E \times (\approx 1))$ = $O(E \log E) = O(E \log V^2) = O(2 \times E \log V) = O(E \log V)$.

```
// inside int main()
  vector< pair<int, ii> > EdgeList;    // (weight, two vertices) of the edge
  for (int i = 0; i < E; i++) {
    scanf("%d %d %d", &u, &v, &w);            // read the triple: (u, v, w)
    EdgeList.push_back(make_pair(w, ii(u, v))); }      // (w, u, v)
  sort(EdgeList.begin(), EdgeList.end()); // sort by edge weight O(E log E)
                     // note: pair object has built-in comparison function
  int mst_cost = 0;
  UnionFind UF(V);                      // all V are disjoint sets initially
  for (int i = 0; i < E; i++) {                         // for each edge, O(E)
    pair<int, ii> front = EdgeList[i];
    if (!UF.isSameSet(front.second.first, front.second.second)) {  // check
      mst_cost += front.first;                  // add the weight of e to MST
      UF.unionSet(front.second.first, front.second.second);    // link them
  } }                        // note: the runtime cost of UFDS is very light
  // note: the number of disjoint sets must eventually be 1 for a valid MST
  printf("MST cost = %d (Kruskal's)\n", mst_cost);
```

Figure 4.11 shows step by step execution of Kruskal's algorithm on the graph shown in Figure 4.10—leftmost. Notice that the final MST is not unique.
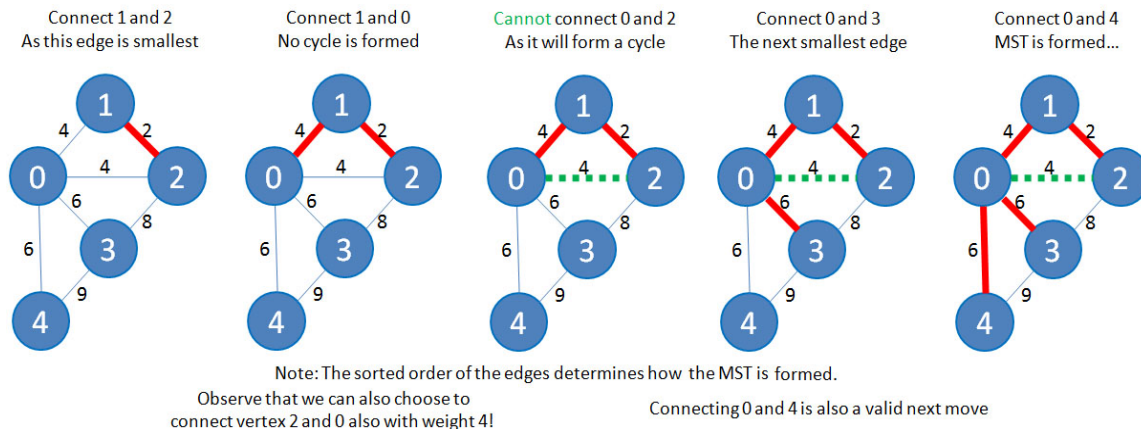
Figure 4.11: Animation of Kruskal's Algorithm for an MST Problem

Exercise 4.3.2.1: The code above only stops after the last edge in EdgeList is processed. In many cases, we can stop Kruskal's *earlier*. Modify the code to implement this!

Exercise 4.3.2.2*: Can you solve the MST problem *faster* than $O(E \log V)$ if the input graph is guaranteed to have edge weights that lie between a small integer range of [0..100]? Is the potential speed-up significant?

4.3.3 Prim's Algorithm

Robert Clay *Prim*'s algorithm first takes a starting vertex (for simplicity, we take vertex 0), flags it as 'taken', and enqueues a pair of information into a priority queue: The weight w and the other end point u of the edge $0 \to u$ that is not taken yet. These pairs are sorted in the priority queue based on increasing weight, and if tie, by increasing vertex number. Then, Prim's algorithm *greedily* selects the pair (w, u) in front of the priority

queue—which has the minimum weight w—if the end point of this edge—which is u—has not been taken before. This is to prevent cycle. If this pair (w, u) is valid, then the weight w is added into the MST cost, u is marked as taken, and pair (w', v) of each edge $u \to v$ with weight w' that is incident to u is enqueued into the priority queue if v has not been taken before. This process is repeated until the priority queue is empty. The code length is about the same as Kruskal's and also runs in $O(\text{process each edge once} \times \text{cost of enqueue/dequeue}) = O(E \times \log E) = O(E \log V)$.

```
vi taken;                        // global boolean flag to avoid cycle
priority_queue<ii> pq;        // priority queue to help choose shorter edges
          // note: default setting for C++ STL priority_queue is a max heap
void process(int vtx) {    // so, we use -ve sign to reverse the sort order
  taken[vtx] = 1;
  for (int j = 0; j < (int)AdjList[vtx].size(); j++) {
    ii v = AdjList[vtx][j];
    if (!taken[v.first]) pq.push(ii(-v.second, -v.first));
} }                              // sort by (inc) weight then by (inc) id

// inside int main()---assume the graph is stored in AdjList, pq is empty
  taken.assign(V, 0);                    // no vertex is taken at the beginning
  process(0);    // take vertex 0 and process all edges incident to vertex 0
  mst_cost = 0;
  while (!pq.empty()) {  // repeat until V vertices (E=V-1 edges) are taken
    ii front = pq.top(); pq.pop();
    u = -front.second, w = -front.first;  // negate the id and weight again
    if (!taken[u])                    // we have not connected this vertex yet
      mst_cost += w, process(u); // take u, process all edges incident to u
  }                                        // each edge is in pq only once!
  printf("MST cost = %d (Prim's)\n", mst_cost);
```

Figure 4.12 shows the step by step execution of Prim's algorithm on the same graph shown in Figure 4.10—leftmost. Please compare it with Figure 4.11 to study the similarities and differences between Kruskal's and Prim's algorithms.
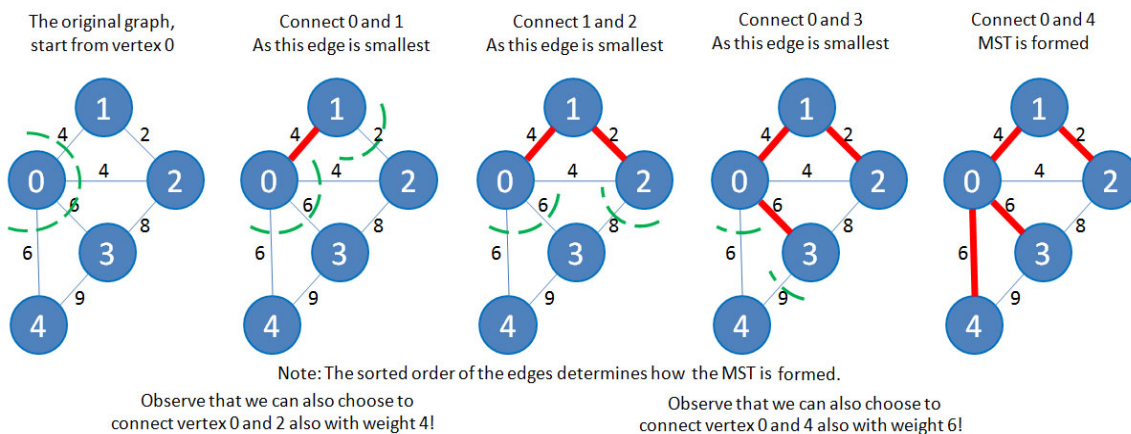
Figure 4.12: Animation of Prim's Algorithm for the same graph as in Figure 4.10—left

Visualization: www.comp.nus.edu.sg/~stevenha/visualization/mst.html

Source code: ch4_03_kruskal_prim.cpp/java

4.3.4   Other Applications

Variants of basic MST problem are interesting. In this section, we will explore some of them.
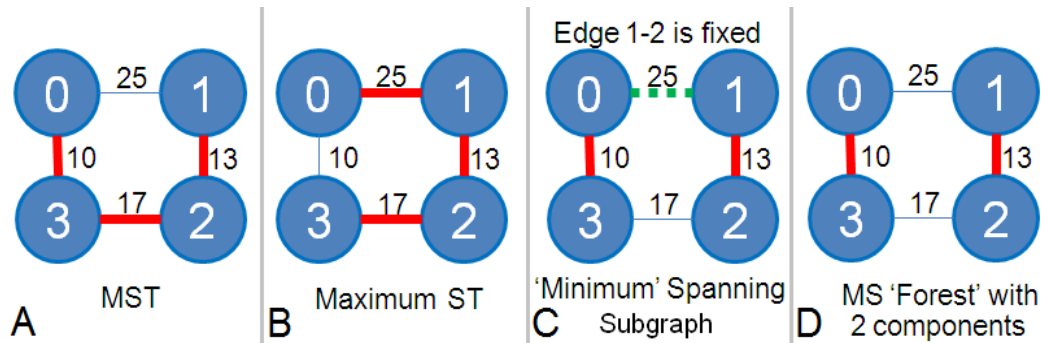
Figure 4.13: From left to right: MST, 'Maximum' ST, 'Minimum' SS, MS 'Forest'

'Maximum' Spanning Tree

This is a simple variant where we want the maximum instead of the minimum ST, for example: UVa 1234 - RACING (note that this problem is written in such a way that it does not look like an MST problem). In Figure 4.13.B, we see an example of a Maximum ST. Compare it with the corresponding MST (Figure 4.13.A).

The solution for this variant is very simple: Modify Kruskal's algorithm a bit, we now simply sort the edges based on *non increasing* weight.

'Minimum' Spanning Subgraph

In this variant, we do not start with a clean slate. Some edges in the given graph have already been fixed and must be taken as part of the solution, for example: UVa 10147 - Highways. These default edges may form a non-tree in the first place. Our task is to continue selecting the remaining edges (if necessary) to make the graph connected in the least cost way. The resulting Spanning Subgraph may not be a tree and even if it is a tree, it may not be the MST. That's why we put the term 'Minimum' in quotes and use the term 'subgraph' rather than 'tree'. In Figure 4.13.C, we see an example when one edge 0-1 is already fixed. The actual MST is 10+13+17 = 40 which omits the edge 0-1 (Figure 4.13.A). However, the solution for this example must be (25)+10+13 = 48 which uses the edge 0-1.

The solution for this variant is simple. After taking into account all the fixed edges and their costs, we continue running Kruskal's algorithm on the remaining free edges until we have a spanning subgraph (or spanning tree).

Minimum 'Spanning Forest'

In this variant, we want to form a forest of K connected components (K subtrees) in the least cost way where K is given beforehand in the problem description, for example: UVa 10369 - Arctic Networks. In Figure 4.13.A, we observe that the MST for this graph is 10+13+17 = 40. But if we are happy with a spanning forest with 2 connected components, then the solution is just 10+13 = 23 on Figure 4.13.D. That is, we omit the edge 2-3 with weight 17 which will connect these two components into one spanning tree if taken.

To get the minimum spanning forest is simple. Run Kruskal's algorithm as per normal, but as soon as the number of connected components equals to the desired pre-determined number K, we can terminate the algorithm.

Second Best Spanning Tree
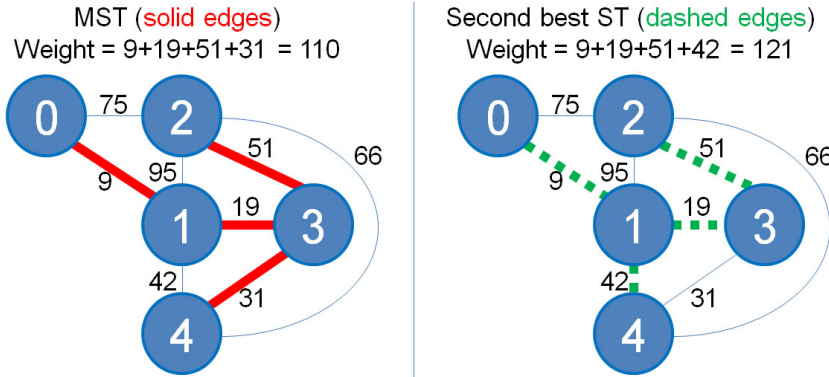
Figure 4.14: Second Best ST (from UVa 10600 [47])

Sometimes, alternative solutions are important. In the context of finding the MST, we may want not just the MST, but also the second best spanning tree, in case the MST is not workable, for example: UVa 10600 - ACM contest and blackout. Figure 4.14 shows the MST (left) and the second best ST (right). We can see that the second best ST is actually the MST with just two edges difference, i.e. one edge is taken out from the MST and another chord[5] edge is added into the MST. Here, edge 3-4 is taken out and edge 1-4 is added in.

A solution for this variant is a modified Kruskal's: Sort the edges in $O(E \log E) = O(E \log V)$, then find the MST using Kruskal's in $O(E)$. Next, for each edge in the MST (there are at most V-1 edges in the MST), temporarily flag it so that it cannot be chosen, then try to find the MST again in $O(E)$ but now *excluding* that flagged edge. Note that we do not have to re-sort the edges at this point. The best spanning tree found after this process is the second best ST. Figure 4.15 shows this algorithm on the given graph. In overall, this algorithm runs in $O(\text{sort the edges once + find the original MST + find the second best ST}) = O(E \log V + E + VE) = O(VE)$.
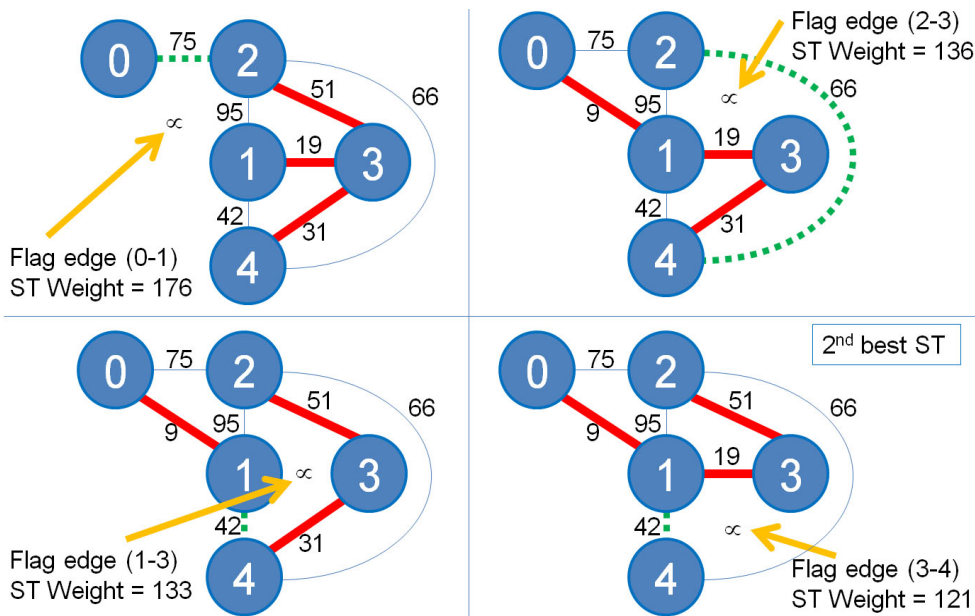
Figure 4.15: Finding the Second Best Spanning Tree from the MST

---

[5]A chord edge is defined as an edge in graph G that is not selected in the MST of G.

Minimax (and Maximin)
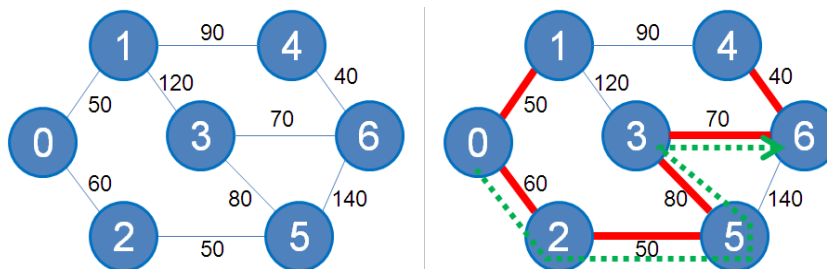
Figure 4.16: Minimax (UVa 10048 [47])

The minimax path problem is a problem of finding the minimum of maximum edge weight among all possible paths between two vertices i to j. The cost for a path from i to j is determined by the maximum edge weight along this path. Among all these possible paths from i to j, pick the one with the minimum max-edge-weight. The reverse problem of maximin is defined similarly.

The minimax path problem between vertex i and j can be solved by modeling it as an MST problem. With a rationale that the problem prefers a path with low individual edge weights even if the path is longer in terms of number of vertices/edges involved, then having the MST (using Kruskal's or Prim's) of the given weighted graph is a correct step. The MST is connected thus ensuring a path between any pair of vertices. The minimax path solution is thus the max edge weight along the unique path between vertex i and j in this MST.

The overall time complexity is $O(\text{build MST} + \text{one traversal on the resulting tree})$. As $E = V - 1$ in a tree, any traversal on tree is just $O(V)$. Thus the complexity of this approach is $O(E \log V + V) = O(E \log V)$.

Figure 4.16—left is a sample test case of UVa 10048 - Audiophobia. We have a graph with 7 vertices and 9 edges. The 6 chosen edges of the MST are shown as thick lines in Figure 4.16, right. Now, if we are asked to find the minimax path between vertex 0 and 6 in Figure 4.16, right, we simply traverse the MST from vertex 0 to 6. There will only be one way, path: 0-2-5-3-6. The maximum edge weight found along the path is the required minimax cost: 80 (due to edge 5-3).

Exercise 4.3.4.1: Solve the five MST problem variants above using Prim's algorithm instead. Which variant(s) is/are not Prim's-friendly?

Exercise 4.3.4.2*: There are better solutions for the Second Best ST problem shown above. Solve this problem with a solution that is better than $O(VE)$. Hints: You can use either Lowest Common Ancestor (LCA) or Union-Find Disjoint-Sets.

Remarks About MST in Programming Contests

To solve many MST problems in today's programming contests, we can rely on Kruskal's algorithm alone and skip Prim's (or other MST) algorithm. Kruskal's is by our reckoning the best algorithm to solve programming contest problems involving MST. It is easy to understand and links well with the Union-Find Disjoint Sets data structure (see Section 2.4.2) that is used to check for cycles. However, as we do love choices, we also include the discussion of the other popular algorithm for MST: Prim's algorithm.

The default (and the most common) usage of Kruskal's (or Prim's) algorithm is to solve the Minimum ST problem (UVa 908, 1174, 1208, 11631), but the easy variant of 'Maximum' ST is also possible (UVa 1234, 10842). Note that most (if not all) MST problems

in programming contests only ask for the *unique* MST cost and not the actual MST itself. This is because there can be different MSTs with the same minimum cost—usually it is too troublesome to write a special checker program to judge such non unique outputs.

The other MST variants discussed in this book like the 'Minimum' Spanning Subgraph (UVa 10147, 10397), Minimum 'Spanning Forest' (UVa 1216, 10369), Second best ST (UVa 10462, 10600), Minimax/Maximin (UVa 534, 544, 10048, 10099) are actually rare.

Nowadays, the more general trend for MST problems is for the problem authors to write the MST problem in such a way that it is not clear that the problem is actually an MST problem (e.g. UVa 1216, 1234, 1235). However, once the contestants spot this, the problem may become 'easy'.

Note that there are harder MST problems that may require more sophisticated algorithm to solve, e.g. Arborescence problem, Steiner tree, degree constrained MST, k-MST, etc.

Programming Exercises related to Minimum Spanning Tree:

- Standard
    1. UVa 00908 - Re-connecting ... (basic MST problem)
    2. *UVa 01174 - IP-TV* (LA 3988, SouthWesternEurope07, MST, classic, just need a mapper to map city names to indices)
    3. UVa 01208 - Oreon (LA 3171, Manila06, MST)
    4. UVa 01235 - Anti Brute Force Lock (LA 4138, Jakarta08, the underlying problem is MST)
    5. UVa 10034 - Freckles (straightforward MST problem)
    6. **UVa 11228 - Transportation System *** (split the output for short versus long edges)
    7. **UVa 11631 - Dark Roads *** (weight of (all graph edges - all MST edges))
    8. UVa 11710 - Expensive Subway (output 'Impossible' if the graph is still disconnected after running MST)
    9. UVa 11733 - Airports (maintain cost at every update)
    10. **UVa 11747 - Heavy Cycle Edges *** (sum the edge weights of the chords)
    11. UVa 11857 - Driving Range (find weight of the last edge added to MST)
    12. IOI 2003 - Trail Maintenance (use efficient incremental MST)
- Variants
    1. UVa 00534 - Frogger (minimax, also solvable with Floyd Warshall's)
    2. UVa 00544 - Heavy Cargo (maximin, also solvable with Floyd Warshall's)
    3. *UVa 01160 - X-Plosives* (count the number of edges not taken by Kruskal's)
    4. UVa 01216 - The Bug Sensor Problem (LA 3678, Kaohsiung06, minimum 'spanning forest')
    5. UVa 01234 - RACING (LA 4110, Singapore07, 'maximum' spanning tree)
    6. **UVa 10048 - Audiophobia *** (minimax, see the discussion above)
    7. UVa 10099 - Tourist Guide (maximin, also solvable with Floyd Warshall's)
    8. UVa 10147 - Highways ('minimum' spanning subgraph)
    9. **UVa 10369 - Arctic Networks *** (minimum spanning 'forest')
    10. UVa 10397 - Connect the Campus ('minimum' spanning subgraph)
    11. UVa 10462 - Is There A Second ... (second best spanning tree)
    12. **UVa 10600 - ACM Contest and ... *** (second best spanning tree)
    13. UVa 10842 - Traffic Flow (find min weighted edge in 'max' spanning tree)

Profile of Algorithm Inventors

Robert Endre Tarjan (born 1948) is an American computer scientist. He is the discoverer of several important graph algorithms. The most important one in the context of competitive programming is the algorithm for finding **Strongly Connected Components algorithm** in a directed graph and the algorithm to find **Articulation Points and Bridges** in an undirected graph (discussed in Section 4.2 together with other DFS variants invented by him and his colleagues [63]). He also invented **Tarjan's off-line Least Common Ancestor algorithm**, invented **Splay Tree data structure**, and analyze the time complexity of the **Union-Find Disjoint Sets data structure** (see Section 2.4.2).

John Edward Hopcroft (born 1939) is an American computer scientist. He is the Professor of Computer Science at Cornell University. Hopcroft received the Turing Award—the most prestigious award in the field and often recognized as the 'Nobel Prize of computing' (jointly with Robert Endre Tarjan in 1986)—for fundamental achievements in the design and analysis of algorithms and data structures. Along with his work with Tarjan on planar graphs (and some other graph algorithms like **finding articulation points/bridges using DFS**) he is also known for the **Hopcroft-Karp's algorithm** for finding matchings in bipartite graphs, invented together with Richard Manning Karp [28] (see Section 9.12).

Joseph Bernard Kruskal, Jr. (1928-2010) was an American computer scientist. His best known work related to competitive programming is the **Kruskal's algorithm** for computing the Minimum Spanning Tree (MST) of a weighted graph. MST have interesting applications in construction and *pricing* of communication networks.

Robert Clay Prim (born 1921) is an American mathematician and computer scientist. In 1957, at Bell Laboratories, he developed Prim's algorithm for solving the MST problem. Prim knows Kruskal as they worked together in Bell Laboratories. Prim's algorithm, was originally discovered earlier in 1930 by Vojtêch Jarník and rediscovered independently by Prim. Thus Prim's algorithm sometimes also known as Jarník-Prim's algorithm.

Vojtêch Jarník (1897-1970) was a Czech mathematician. He developed the graph algorithm now known as Prim's algorithm. In the era of fast and widespread publication of scientific results nowadays. Prim's algorithm would have been credited to Jarník instead of Prim.

Edsger Wybe Dijkstra (1930-2002) was a Dutch computer scientist. One of his famous contributions to computer science is the shortest path-algorithm known as **Dijkstra's algorithm** [10]. He does not like 'GOTO' statement and influenced the widespread deprecation of 'GOTO' and its replacement: structured control constructs. One of his famous Computing phrase: "two or more, use a for".

Richard Ernest Bellman (1920-1984) was an American applied mathematician. Other than inventing the **Bellman Ford's algorithm** for finding shortest paths in graphs that have negative weighted edges (and possibly negative weight cycle), Richard Bellman is more well known by his invention of the *Dynamic Programming* technique in 1953.

Lester Randolph Ford, Jr. (born 1927) is an American mathematician specializing in network flow problems. Ford's 1956 paper with Fulkerson on the maximum flow problem and the **Ford Fulkerson's method** for solving it, established the max-flow min-cut theorem.

Delbert Ray Fulkerson (1924-1976) was a mathematician who co-developed the **Ford Fulkerson's method**, an algorithm to solve the Max Flow problem in networks. In 1956, he published his paper on the Ford Fulkerson's method together with Lester R. Ford.

4.4 Single-Source Shortest Paths

4.4.1 Overview and Motivation

Motivating problem: Given a *weighted* graph G and a starting source vertex *s*, what are the *shortest paths* from *s* to *every other vertices* of G?

This problem is called the *Single-Source*[6] *Shortest Paths* (SSSP) problem on a *weighted graph*. It is a classical problem in graph theory and has many real life applications. For example, we can model the city that we live in as a graph. The vertices are the road junctions. The edges are the roads. The time taken to traverse a road is the weight of the edge. You are currently in one road junction. What is the shortest possible time to reach another certain road junction?

There are efficient algorithms to solve this SSSP problem. If the graph is unweighted (or all edges have equal or constant weight), we can use the efficient $O(V + E)$ BFS algorithm shown earlier in Section 4.2.2. For a general weighted graph, BFS does not work correctly and we should use algorithms like the $O((V + E) \log V)$ Dijkstra's algorithm or the $O(VE)$ Bellman Ford's algorithm. These various algorithms are discussed below.

Exercise 4.4.1.1*: Prove that the shortest path between two vertices *i* and *j* in a graph G that has no negative weight cycle must be a *simple* path (acyclic)!

Exercise 4.4.1.2*: Prove: Subpaths of shortest paths from *u* to *v* are shortest paths!

4.4.2 SSSP on Unweighted Graph

Let's revisit Section 4.2.2. The fact that BFS visits vertices of a graph layer by layer from a source vertex (see Figure 4.3) turns BFS into a natural choice to solve the SSSP problems on *unweighted* graphs. In an unweighted graph, the distance between two neighboring vertices connected with an edge is simply one unit. Therefore, the layer count of a vertex that we have seen in Section 4.2.2 is precisely the shortest path length from the source to that vertex. For example in Figure 4.3, the shortest path from vertex 5 to vertex 7, is 4, as 7 is in the fourth layer in BFS sequence of visitation starting from vertex 5.

Some programming problems require us to *reconstruct* the actual shortest path, not just the shortest path length. For example, in Figure 4.3, the shortest path from 5 to 7 is $5 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 7$. Such reconstruction is easy if we store the shortest path (actually BFS) spanning tree[7]. This can be easily done using vector of integers `vi p`. Each vertex *v* remembers its parent *u* (`p[v] = u`) in the shortest path spanning tree. For this example, vertex 7 remembers 3 as its parent, vertex 3 remembers 2, vertex 2 remembers 1, vertex 1 remembers 5 (the source). To reconstruct the actual shortest path, we can do a simple recursion from the last vertex 7 until we hit the source vertex 5. The modified BFS code (check the comments) is relatively simple:

```
void printPath(int u) {                    // extract information from 'vi p'
  if (u == s) { printf("%d", s); return; }    // base case, at the source s
  printPath(p[u]);    // recursive: to make the output format: s -> ... -> t
  printf(" %d", u); }
```

---

[6]This generic SSSP problem can also be used to solve the: 1). Single-Pair (or Single-Source Single-Destination) SP problem where both source + destination vertices are given and 2). Single-Destination SP problem where we just reverse the role of source/destination vertices.

[7]Reconstructing the shortest path is not shown in the next two subsections (Dijkstra's/Bellman Ford's) but the idea is the same as the one shown here (and with reconstructing DP solution in Section 3.5.1).

```
// inside int main()
  vi dist(V, INF); dist[s] = 0;          // distance from source s to s is 0
  queue<int> q; q.push(s);
  vi p;                                   // addition: the predecessor/parent vector
  while (!q.empty()) {
    int u = q.front(); q.pop();
    for (int j = 0; j < (int)AdjList[u].size(); j++) {
      ii v = AdjList[u][j];
      if (dist[v.first] == INF) {
        dist[v.first] = dist[u] + 1;
        p[v.first] = u;       // addition: the parent of vertex v.first is u
        q.push(v.first);
} } }
  printPath(t), printf("\n");     // addition: call printPath from vertex t
```

Source code: `ch4_04_bfs.cpp/java`

We would like to remark that recent programming contest problems involving BFS are no longer written as straightforward SSSP problems but written in a much more creative fashion. Possible variants include: BFS on implicit graph (2D grid: UVa 10653 or 3-D grid: UVa 532), BFS with the printing of the actual shortest path (UVa 11049), BFS on graph with some blocked vertices (UVa 10977), BFS from multi-sources (UVa 11101, 11624), BFS with single destination—solved by reversing the role of source and destination (UVa 11513), BFS with non-trivial states (UVa 10150)—more such problems in Section 8.2.3, etc. Since there are many interesting variants of BFS, we recommend that the readers try to solve as many problems as possible from the programming exercises listed in this section.

Exercise 4.4.2.1: We can run BFS from > 1 sources. We call this variant the Multi-Sources Shortest Paths (MSSP) on unweighted graph problem. Try solving UVa 11101 and 11624 to get the idea of MSSP on unweighted graph. A naïve solution is to call BFS multiple times. If there are k possible sources, such solution will run in $O(k \times (V + E))$. Can you do better?

Exercise 4.4.2.2: Suggest a simple improvement to the given BFS code above if you are asked to solve the Single-Source *Single-Destination* Shortest Path problem on an unweighted graph. That's it, you are given both the source *and* the destination vertex.

Exercise 4.4.2.3: Explain the reason why we can use BFS to solve an SSSP problem on a weighted graph where *all edges* has the same weight C?

Exercise 4.4.2.4*: Given an $R \times C$ grid map like shown below, determine the shortest path from any cell labeled as 'A' to any cell labeled as 'B'. You can only walk through cells labeled with '.' in NESW direction (counted as *one* unit) and cells labeled with alphabet 'A'-'Z' (counted as *zero* unit)! Can you solve this in $O(R \times C)$?

```
....................CCCC.      // The answer for this test case is 13 units
AAAAA...............CCCC.                 // Solution: Walk east from
AAAAA.AAA...........CCCC.     // the rightmost A to leftmost C in this row
AAAAAAAAA....###....CCCC.   // then walk south from rightmost C in this row
AAAAAAAAA...............                                         // down
AAAAAAAAA...............                                           // to
.......DD..............BB            // the leftmost B in this row
```

4.4.3 SSSP on Weighted Graph

If the given graph is *weighted*, BFS does not work. This is because there can be 'longer' path(s) (in terms of number of vertices and edges involved in the path) but has smaller total weight than the 'shorter' path found by BFS. For example, in Figure 4.17, the shortest path from source vertex 2 to vertex 3 is not via direct edge $2 \to 3$ with weight 7 that is normally found by BFS, but a 'detour' path: $2 \to 1 \to 3$ with smaller total weight $2 + 3 = 5$.

To solve the SSSP problem on weighted graph, we use a *greedy* Edsger Wybe *Dijkstra*'s algorithm. There are several ways to implement this classic algorithm. In fact, Dijkstra's original paper that describes this algorithm [10] does not describe a specific implementation. Many other Computer Scientists proposed implementation variants based on Dijkstra's original work. Here we adopt one of the easiest implementation variant that uses *built-in* C++ STL `priority_queue` (or Java `PriorityQueue`). This is to keep the length of code *minimal*—a necessary feature in competitive programming.

This `pq` only contains one item initially: The base case $(0, s)$ which is true for the source vertex. Then, this Dijkstra's implementation variant repeats the following process until `pq` is empty: It greedily takes out vertex information pair (d, u) from the front of `pq`. If the distance to u from source recorded in d greater than `dist[u]`, it ignores u; otherwise, it process u. The reason for this special check is shown below.

This Dijkstra's variant maintains a **priority** queue called `pq` that stores pairs of vertex information. The first and the second item of the pair is the distance of the vertex from the source and the vertex number, respectively. This `pq` is sorted based on *increasing distance* from the source, and if tie, by vertex number. This is different from another Dijkstra's implementation that uses binary heap feature that is not supported in built-in library[8].

When this algorithm process u, it tries to relax[9] all neighbors v of u. Every time it relaxes an edge $u \to v$, it will *enqueue* a pair (newer/shorter distance to v from source, v) into `pq` and *leave the inferior pair* (older/longer distance to v from source, v) inside `pq`. This is called 'Lazy Deletion' and it causes *more than one copy* of the same vertex in `pq` with *different distances* from source. That is why we have the check earlier to process only the *first dequeued* vertex information pair which has the correct/shorter distance (other copies will have the outdated/longer distance). The code is shown below and it looks very similar to BFS and Prim's code shown in Section 4.2.2 and 4.3.3, respectively.

```
vi dist(V, INF); dist[s] = 0;                  // INF = 1B to avoid overflow
priority_queue< ii, vector<ii>, greater<ii> > pq; pq.push(ii(0, s));
while (!pq.empty()) {                                            // main loop
  ii front = pq.top(); pq.pop(); // greedy: get shortest unvisited vertex
  int d = front.first, u = front.second;
  if (d > dist[u]) continue;              // this is a very important check
  for (int j = 0; j < (int)AdjList[u].size(); j++) {
    ii v = AdjList[u][j];                        // all outgoing edges from u
    if (dist[u] + v.second < dist[v.first]) {
      dist[v.first] = dist[u] + v.second;             // relax operation
      pq.push(ii(dist[v.first], v.first));
} } }     // this variant can cause duplicate items in the priority queue
```

Source code: `ch4_05_dijkstra.cpp/java`

---

[8]The usual implementation of Dijkstra's (e.g. see [7, 38, 8]) requires `heapDecreaseKey` operation in binary heap DS that is not supported by built-in priority queue in C++ STL or Java API. Dijkstra's implementation variant discussed in this section uses only two basic priority queue operations: `enqueue` and `dequeue`.

[9]The operation: `relax(u, v, w_u_v)` sets `dist[v] = min(dist[v], dist[u] + w_u_v)`.

In Figure 4.17, we show a step by step example of running this Dijkstra's implementation variant on a small graph and $s = 2$. Take a careful look at the content of `pq` at each step.
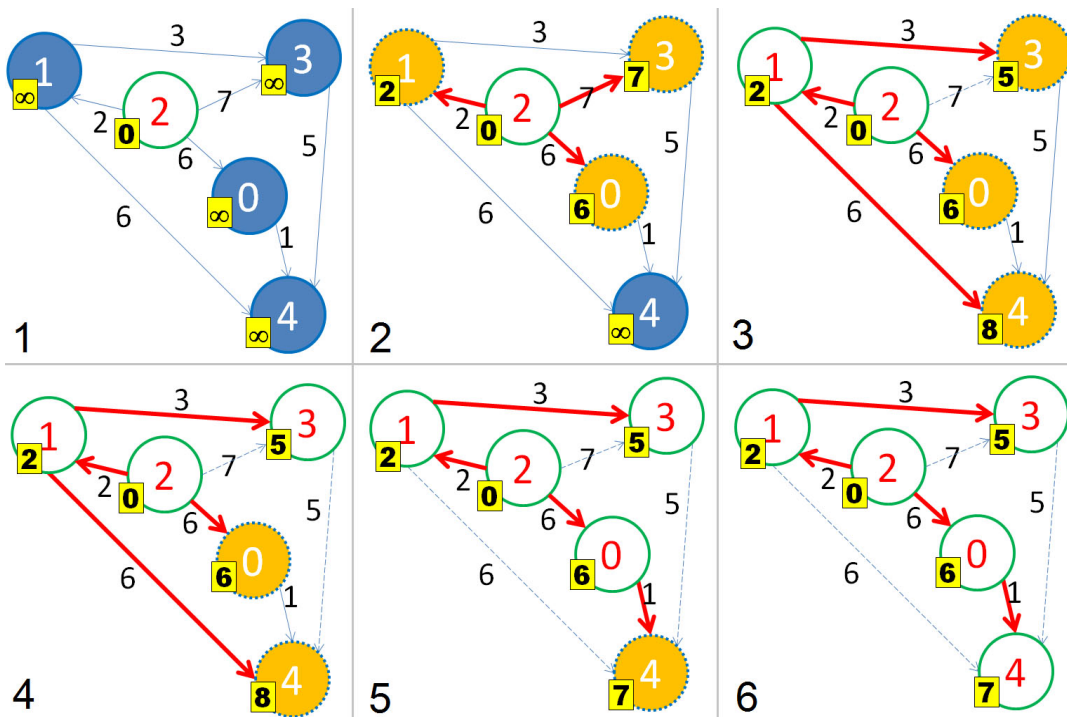
Figure 4.17: Dijkstra Animation on a Weighted Graph (from UVa 341 [47])

1. At the beginning, only `dist[s]` = `dist[2]` = 0, `priority_queue` pq is $\{(0,2)\}$.

2. Dequeue vertex information pair $(0,2)$ from pq. Relax edges incident to vertex 2 to get `dist[0]` = 6, `dist[1]` = 2, and `dist[3]` = 7. Now pq contains $\{(2,1), (6,0), (7,3)\}$.

3. Among the unprocessed pairs in pq, $(2,1)$ is in the front of pq. We dequeue $(2,1)$ and relax edges incident to vertex 1 to get `dist[3]` = `min(dist[3], dist[1]` + `weight(1,3))` = `min(7, 2+3)` = 5 and `dist[4]` = 8. Now pq contains $\{(5,3), (6,0), (7,3), (8,4)\}$. See that we have 2 entries of vertex 3 in our pq with increasing distance from source s. We do not immediately delete the inferior pair $(7,3)$ from the pq and rely on future iterations of our Dijkstra's variant to correctly pick the one with minimal distance later, which is pair $(5,3)$. This is called 'lazy deletion'.

4. We dequeue $(5,3)$ and try to do `relax(3,4,5)`, i.e. $5+5 = 10$. But `dist[4]` = 8 (from path 2-1-4), so `dist[4]` is unchanged. Now pq contains $\{(6,0), (7,3), (8,4)\}$.

5. We dequeue $(6,0)$ and do `relax(0,4,1)`, making `dist[4]` = 7 (the shorter path from 2 to 4 is now 2-0-4 instead of 2-1-4). Now pq contains $\{(7,3), (7,4), (8,4)\}$ with 2 entries of vertex 4. This is another case of 'lazy deletion'.

6. Now, $(7,3)$ can be ignored as we know that its `d` > `dist[3]` (i.e. $7 > 5$). This iteration 6 is where the actual deletion of the inferior pair $(7,3)$ is executed rather than iteration 3 earlier. By deferring it until iteration 6, the inferior pair $(7,3)$ is now located in the easy position for the standard $O(\log n)$ deletion in the min heap: At the root of the min heap, i.e. the front of the priority queue.

7. Then $(7,4)$ is processed as before but nothing change. Now pq contains only $\{(8,4)\}$.

8. Finally $(8,4)$ is ignored again as its `d` > `dist[4]` (i.e. $8 > 7$). This Dijkstra's implementation variant stops here as the pq is now empty.

Sample Application: UVa 11367 - Full Tank?

Abridged problem description: Given a connected weighted graph *length* that stores the road length between E pairs of cities i and j ($1 \leq V \leq 1000, 0 \leq E \leq 10000$), the price $p[i]$ of fuel at each city i, and the fuel tank capacity c of a car ($1 \leq c \leq 100$), determine the cheapest trip cost from starting city s to ending city e using a car with fuel capacity c. All cars use one unit of fuel per unit of distance and start with an empty fuel tank.

With this problem, we want to discuss the importance of *graph modeling*. The explicitly given graph in this problem is a weighted graph of the road network. However, we cannot solve this problem with just this graph. This is because the state[10] of this problem requires not just the current location (city) but also the fuel level at that location. Otherwise, we cannot determine whether the car has enough fuel to make a trip along a certain road (because we cannot refuel in the middle of the road). Therefore, we use a pair of information to represent the state: (*location*, *fuel*) and by doing so, the total number of vertices of the modified graph *explodes* from just 1000 vertices to $1000 \times 100 = 100000$ vertices. We call the modified graph: 'State-Space' graph.

In the State-Space graph, the source vertex is state $(s, 0)$—at starting city s with empty fuel tank and the target vertices are states (e, any)—at ending city e with any level of fuel between $[0..c]$. There are two types of edge in the State-Space graph: 0-weighted edge that goes from vertex $(x, fuel_x)$ to vertex $(y, fuel_x - length(x, y))$ if the car has sufficient fuel to travel from vertex x to vertex y, and the $p[x]$-weighted edge that goes from vertex $(x, fuel_x)$ to vertex $(x, fuel_x + 1)$ if the car can refuel at vertex x by one unit of fuel (note that the fuel level cannot exceed the fuel tank capacity c). Now, running Dijkstra's on this State-Space graph gives us the solution for this problem (also see Section 8.2.3 for further discussions).

Exercise 4.4.3.1: The modified Dijkstra's implementation variant above may be different from what you learn from other books (e.g. [7, 38, 8]). Analyze if this variant still runs in $O((V + E) \log V)$ on various types of weighted graphs (also see the next **Exercise 4.4.3.2***)?

Exercise 4.4.3.2*: Construct a graph that has negative weight edges but no negative cycle that can significantly slow down this Dijkstra's implementation!

Exercise 4.4.3.3: The sole reason why this variant allows duplicate vertices in the priority queue is so that it can use built-in priority queue library as it is. There is another alternative implementation variant that also has minimal coding. It uses `set`. Implement this variant!

Exercise 4.4.3.4: The source code shown above uses `priority_queue< ii, vector<ii>, greater<ii> > pq;` to sort pairs of integers by increasing distance from source s. How can we achieve the same effect without defining comparison operator for the `priority_queue`? Hint: We have used similar trick with Kruskal's algorithm implementation in Section 4.3.2.

Exercise 4.4.3.5: In **Exercise 4.4.2.2**, we have seen a way to speed up the solution of a shortest paths problem if you are given both the source and the destination vertices. Can the same speedup trick be used for all kinds of weighted graph?

Exercise 4.4.3.6: The graph modeling for UVa 11367 above transform the SSSP problem on weighted graph into SSSP problem on weighted *State-Space* graph. Can we solve this problem with DP? If we can, why? If we cannot, why not? Hint: Read Section 4.7.1.

---

[10]Recall: State is a subset of parameters of the problem that can succinctly describes the problem.

4.4.4   SSSP on Graph with Negative Weight Cycle

If the input graph has negative edge weight, typical Dijkstra's implementation (e.g. [7, 38, 8]) can produces wrong answer. However, Dijkstra's implementation variant shown in Section 4.4.3 above will work just fine, albeit slower. Try it on the graph in Figure 4.18.

This is because Dijkstra's implementation variant will keep inserting new vertex information pair into the priority queue every time it does a relax operation. So, if the graph has no negative weight *cycle*, the variant will keep propagating the shortest path distance information until there is no more possible relaxation (which implies that all shortest paths from the source have been found). However, when given a graph with negative weight *cycle*, the variant—if implemented as shown in Section 4.4.3 above—will be trapped in an infinite loop.
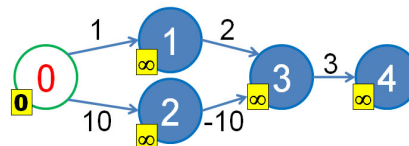
Figure 4.18: -ve Weight

Example: See the graph in Figure 4.19. Path 1-2-1 is a negative cycle. The weight of this cycle is 15 + (-42) = -27.

To solve the SSSP problem in the potential presence of negative weight *cycle(s)*, the more generic (but slower) Bellman Ford's algorithm must be used. This algorithm was invented by Richard Ernest *Bellman* (the pioneer of DP techniques) and Lester Randolph *Ford*, Jr (the same person who invented Ford Fulkerson's method in Section 4.6.2). The main idea of this algorithm is simple: Relax all E edges (in arbitrary order) V-1 times!

Initially `dist[s] = 0`, the base case. If we relax an edge $s \rightarrow u$, then `dist[u]` will have the correct value. If we then relax an edge $u \rightarrow v$, then `dist[v]` will also have the correct value. If we have relaxed all E edges V-1 times, then the shortest path from the source vertex to the furthest vertex from the source (which will be a simple path with V-1 edges) should have been correctly computed. The main part of Bellman Ford's code is simpler than BFS and Dijsktra's code:

```
vi dist(V, INF); dist[s] = 0;
for (int i = 0; i < V - 1; i++)            // relax all E edges V-1 times
  for (int u = 0; u < V; u++)     // these two loops = O(E), overall O(VE)
    for (int j = 0; j < (int)AdjList[u].size(); j++) {
      ii v = AdjList[u][j];              // record SP spanning here if needed
      dist[v.first] = min(dist[v.first], dist[u] + v.second);     // relax
    }
```

The complexity of Bellman Ford's algorithm is $O(V^3)$ if the graph is stored as an Adjacency Matrix or $O(VE)$ if the graph is stored as an Adjacency List. This is simply because if we use Adjacency Matrix, we need $O(V^2)$ to enumerate all the edges in our graph. Both time complexities are (much) slower compared to Dijkstra's. However, the way Bellman Ford's works ensure that it will never be trapped in an infinite loop even if the given graph has negative cycle. In fact, Bellman Ford's algorithm can be used to detect *the presence* of negative cycle (e.g. UVa 558 - Wormholes) although such SSSP problem is ill-defined.
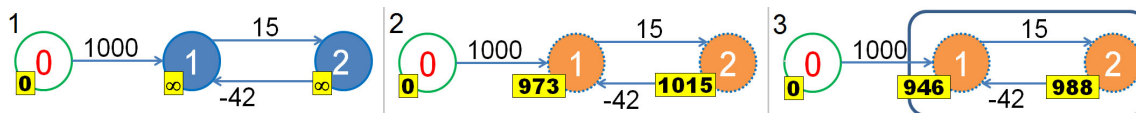
Figure 4.19: Bellman Ford's can detect the presence of negative cycle (from UVa 558 [47])

151

In **Exercise 4.4.4.1**, we prove that after relaxing all E edges V-1 times, the SSSP problem should have been solved, i.e. we cannot relax any more edge. As the corollary: If we can still relax an edge, there must be a negative cycle in our weighted graph.

For example, in Figure 4.19—left, we see a simple graph with a negative cycle. After 1 pass, `dist[1]` = 973 and `dist[2]` = 1015 (middle). After V-1 = 2 passes, `dist[1]` = 946 and `dist[2]` = 988 (right). As there is a negative cycle, we can still do this again (and again), i.e. we can still relax `dist[2]` = 946+15 = 961. This is lower than the current value of `dist[2]` = 988. The presence of a negative cycle causes the vertices reachable from this negative cycle to have ill-defined shortest paths information. This is because one can simply traverse this negative cycle infinite number of times to make all reachable vertices from this negative cycle to have negative infinity shortest paths information. The code to check for negative cycle is simple:

```
// after running the O(VE) Bellman Ford's algorithm shown above
  bool hasNegativeCycle = false;
  for (int u = 0; u < V; u++)                        // one more pass to check
    for (int j = 0; j < (int)AdjList[u].size(); j++) {
      ii v = AdjList[u][j];
      if (dist[v.first] > dist[u] + v.second)  // if this is still possible
        hasNegativeCycle = true;               // then negative cycle exists!
    }
  printf("Negative Cycle Exist? %s\n", hasNegativeCycle ? "Yes" : "No");
```

In programming contests, the slowness of Bellman Ford's and its negative cycle detection feature causes it to be used only to solve the SSSP problem on *small* graph which is *not guaranteed* to be free from negative weight cycle.

Exercise 4.4.4.1: Why just by relaxing all E edges of our weighted graph $V - 1$ times, we will have the correct SSSP information? Prove it!

Exercise 4.4.4.2: The worst case time complexity of $O(VE)$ is too large in practice. For most cases, we can actually stop Bellman Ford's (much) earlier. Suggest a simple improvement to the given code above to make Bellman Ford's *usually* runs faster than $O(VE)$!

Exercise 4.4.4.3*: A known improvement for Bellman Ford's (especially among Chinese programmers) is the SPFA (Shortest Path Faster Algorithm). Study Section 9.30!

Visualization: `www.comp.nus.edu.sg/~stevenha/visualization/sssp.html`

Source code: `ch4_06_bellman_ford.cpp/java`

Programming Exercises related to Single-Source Shortest Paths:

- On Unweighted Graph: BFS, Easier

  1. UVa 00336 - A Node Too Far (discussed in this section)
  2. UVa 00383 - Shipping Routes (simple SSSP solvable with BFS, use mapper)
  3. *UVa 00388 - Galactic Import* (key idea: we want to minimize planet movements because every edge used decreases value by 5%)
  4. **UVa 00429 - Word Transformation *** (each word is a vertex, connect 2 words with an edge if differ by 1 letter)
  5. UVa 00627 - The Net (also print the path, see discussion in this section)
  6. UVa 00762 - We Ship Cheap (simple SSSP solvable with BFS, use mapper)
  7. **UVa 00924 - Spreading the News *** (the spread is like BFS traversal)
  8. *UVa 01148 - The mysterious X network* (LA 3502, SouthWesternEurope05, single source, single target, shortest path problem but exclude endpoints)
  9. UVa 10009 - All Roads Lead Where? (simple SSSP solvable with BFS)
  10. UVa 10422 - Knights in FEN (solvable with BFS)
  11. UVa 10610 - Gopher and Hawks (solvable with BFS)
  12. **UVa 10653 - Bombs; NO they ... *** (efficient BFS implementation)
  13. UVa 10959 - The Party, Part I (SSSP from source 0 to the rest)

- On Unweighted Graph: BFS, Harder

  1. **UVa 00314 - Robot *** (state: (position, direction), transform input graph)
  2. UVa 00532 - Dungeon Master (3-D BFS)
  3. *UVa 00859 - Chinese Checkers* (BFS)
  4. *UVa 00949 - Getaway* (interesting graph data structure twist)
  5. UVa 10044 - Erdos numbers (the input parsing part is troublesome; if you encounter difficulties with this, see Section 6.2)
  6. UVa 10067 - Playing with Wheels (implicit graph in problem statement)
  7. UVa 10150 - Doublets (BFS state is string!)
  8. UVa 10977 - Enchanted Forest (BFS with blocked states)
  9. UVa 11049 - Basic Wall Maze (some restricted moves, print the path)
  10. **UVa 11101 - Mall Mania *** (multi-sources BFS from m1, get minimum at border of m2)
  11. UVa 11352 - Crazy King (filter the graph first, then it becomes SSSP)
  12. UVa 11624 - Fire (multi-sources BFS)
  13. UVa 11792 - Krochanska is Here (be careful with the 'important station')
  14. **UVa 12160 - Unlock the Lock *** (LA 4408, KualaLumpur08, Vertices = The numbers; Link two numbers with an edge if we can use button push to transform one into another; use BFS to get the answer)

- On Weighted Graph: Dijkstra's, Easier

  1. **UVa 00929 - Number Maze *** (on a 2D maze/implicit graph)
  2. ***UVa 01112 - Mice and Maze **** (LA 2425, SouthwesternEurope01, run Dijkstra's from destination)
  3. UVa 10389 - Subway (use basic geometry skill to build the weighted graph, then run Dijkstra's)
  4. **UVa 10986 - Sending email *** (straightforward Dijkstra's application)

- On Weighted Graph: Dijkstra's, Harder

  1. UVa 01202 - Finding Nemo (LA 3133, Beijing04, SSSP, Dijkstra's on grid: treat each cell as a vertex; the idea is simple but one should be careful with the implementation)

  2. UVa 10166 - Travel (this can be modeled as a shortest paths problem)

  3. *UVa 10187 - From Dusk Till Dawn* (special cases: start = destination: 0 litre; starting or destination city not found or destination city not reachable from starting city: no route; the rest: Dijkstra's)

  4. UVa 10278 - Fire Station (Dijkstra's from fire stations to all intersections; need pruning to pass the time limit)

  5. *UVa 10356 - Rough Roads* (we can attach one extra information to each vertex: whether we come to that vertex using cycle or not; then, run Dijkstra's to solve SSSP on this modified graph)

  6. UVa 10603 - Fill (state: (a, b, c), source: (0, 0, c), 6 possible transitions)

  7. **UVa 10801 - Lift Hopping *** (model the graph carefully!)

  8. *UVa 10967 - The Great Escape* (model the graph; shortest path)

  9. *UVa 11338 - Minefield* (it seems that the test data is weaker than what the problem description says ($n \leq 10000$); we use $O(n^2)$ loop to build the weighted graph and runs Dijkstra's without getting TLE)

  10. UVa 11367 - Full Tank? (discussed in this section)

  11. UVa 11377 - Airport Setup (model the graph carefully: A city to other city with no airport has edge weight 1. A city to other city with airport has edge weight 0. Do Dijkstra's from source. If the start and end city are the same and has no airport, the answer should be 0.)

  12. **UVa 11492 - Babel *** (graph modeling; each word is a vertex; connect two vertices with an edge if they share common language and have different 1st char; connect a source vertex to all words that belong to start language; connect all words that belong to finish language to sink vertex; we can transfer vertex weight to edge weight; then SSSP from source vertex to sink vertex)

  13. UVa 11833 - Route Change (stop Dijkstra's at service route path plus some modification)

  14. ***UVa 12047 - Highest Paid Toll**** (clever usage of Dijkstra's; run Dijkstra's from source and from destination; try all edge (u, v) if $dist[source][u] + weight(u, v) + dist[v][destination] \leq p$; record the largest edge weight found)

  15. *UVa 12144 - Almost Shortest Path* (Dijkstra's; store multiple predecessors)

  16. IOI 2011 - Crocodile (can be modeled as an SSSP problem)

- SSSP on Graph with Negative Weight Cycle (Bellman Ford's)

  1. **UVa 00558 - Wormholes *** (checking the existence of negative cycle)

  2. ***UVa 10449 - Traffic**** (find the minimum weight path, which may be negative; be careful: ∞ + negative weight is lower than ∞!)

  3. **UVa 10557 - XYZZY *** (check 'positive' cycle, check connectedness!)

  4. UVa 11280 - Flying to Fredericton (modified Bellman Ford's)

4.5   All-Pairs Shortest Paths

4.5.1   Overview and Motivation

Motivating Problem: Given a connected, weighted graph G with $V \leq 100$ and two vertices s and d, find the maximum possible value of `dist[s][i] + dist[i][d]` over all possible $i \in [0 \ldots V-1]$. This is the key idea to solve UVa 11463 - Commandos. However, what is the best way to implement the solution code for this problem?

This problem requires the shortest path information from all possible sources (all possible vertices) of G. We can make V calls of Dijkstra's algorithm that we have learned earlier in Section 4.4.3 above. However, can we solve this problem in a *shorter way*—in terms of code length? The answer is yes. If the given weighted graph has $V \leq 400$, then there is another algorithm that is *simpler to code*.

Load the small graph into an Adjacency Matrix and then run the following four-liner code with three nested loops shown below. When it terminates, `AdjMat[i][j]` will contain the shortest path distance between two pair of vertices i and j in G. The original problem (UVa 11463 above) now becomes easy.

```
// inside int main()
  // precondition: AdjMat[i][j] contains the weight of edge (i, j)
  // or INF (1B) if there is no such edge
  // AdjMat is a 32-bit signed integer array
  for (int k = 0; k < V; k++)         // remember that loop order is k->i->j
    for (int i = 0; i < V; i++)
      for (int j = 0; j < V; j++)
        AdjMat[i][j] = min(AdjMat[i][j], AdjMat[i][k] + AdjMat[k][j]);
```

Source code: `ch4_07_floyd_warshall.cpp/java`

This algorithm is called Floyd Warshall's algorithm, invented by Robert W *Floyd* [19] and Stephen *Warshall* [70]. Floyd Warshall's is a DP algorithm that clearly runs in $O(V^3)$ due to its 3 nested loops[11]. Therefore, it can only be used for graph with $V \leq 400$ in programming contest setting. In general, Floyd Warshall's solves another classical graph problem: The All-Pairs Shortest Paths (APSP) problem. It is an alternative algorithm (for small graphs) compared to calling SSSP algorithm multiple times:

1. V calls of $O((V+E) \log V)$ Dijkstra's $= O(V^3 \log V)$ if $E = O(V^2)$.

2. V calls of $O(VE)$ Bellman Ford's $= O(V^4)$ if $E = O(V^2)$.

In programming contest setting, Floyd Warshall's main attractiveness is basically its implementation speed—four short lines only. If the given graph is small ($V \leq 400$), do not hesitate to use this algorithm—even if you only need a solution for the SSSP problem.

Exercise 4.5.1.1: Is there any specific reason why `AdjMat[i][j]` must be set to 1B (10^9) to indicate that there is no edge between 'i' to 'j'? Why don't we use $2^{31} - 1$ (MAX_INT)?

Exercise 4.5.1.2: In Section 4.4.4, we differentiate graph with negative weight edges but no negative cycle and graph with negative cycle. Will this short Floyd Warshall's algorithm works on graph with negative weight and/or negative cycle? Do some experiment!

---

[11]Floyd Warshall's must use Adjacency Matrix so that the weight of edge (i, j) can be accessed in $O(1)$.

4.5.2 Explanation of Floyd Warshall's DP Solution

We provide this section for the benefit of readers who are interested to know why Floyd Warshall's works. This section can be skipped if you just want to use this algorithm per se. However, examining this section can further strengthen your DP skill. Note that there are graph problems that have no classical algorithm yet and must be solved with DP techniques (see Section 4.7.1).
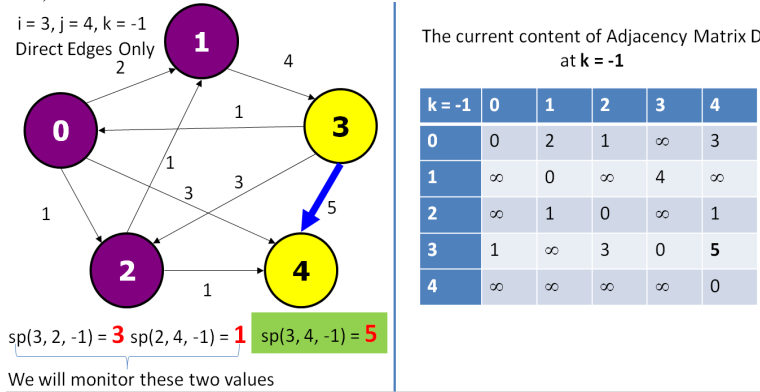
Figure 4.20: Floyd Warshall's Explanation 1

The basic idea behind Floyd Warshall's is to gradually allow the usage of intermediate vertices (vertex [0..k]) to form the shortest paths. We denote the shortest path from vertex i to vertex j using only intermediate vertices [0..k] as sp(i,j,k). Let the vertices be labeled from 0 to V-1. We start with direct edges only when $k = -1$, i.e. sp(i,j,-1) = weight of edge (i, j). Then, we find shortest paths between any two vertices with the help of restricted intermediate vertices from vertex [0..k]. In Figure 4.20, we want to find sp(3,4,4)—the shortest path from vertex 3 to vertex 4, using any intermediate vertex in the graph (vertex [0..4]). The eventual shortest path is path 3-0-2-4 with cost 3. But how to reach this solution? We know that by using only direct edges, sp(3,4,-1) = 5, as shown in Figure 4.20. The solution for sp(3,4,4) will *eventually* be reached from sp(3,2,2)+sp(2,4,2). But with using only direct edges, sp(3,2,-1)+sp(2,4,-1) = 3+1 = 4 is still > 3.
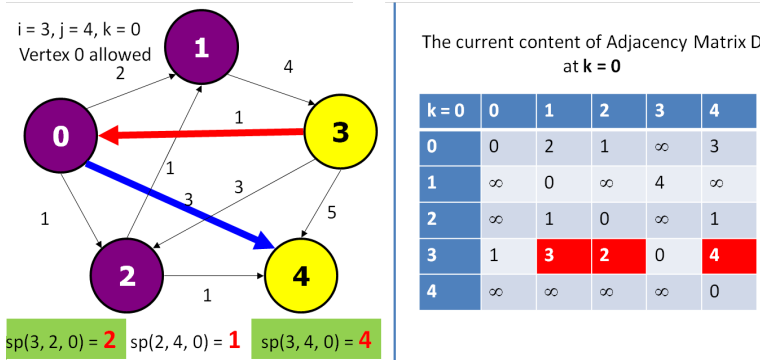
Figure 4.21: Floyd Warshall's Explanation 2

Floyd Warshall's then gradually allow $k = 0$, then $k = 1$, $k = 2$ ..., up to $k = V$-1. When we allow $k = 0$, i.e. vertex 0 can now be used as an intermediate vertex, then sp(3,4,0) is reduced as sp(3,4,0) = sp(3,0,-1) + sp(0,4,-1) = 1+3 = 4, as shown in Figure 4.21. Note that with $k = 0$, sp(3,2,0)—which we will need later—also drop from 3 to sp(3,0,-1) + sp(0,2,-1) = 1+1 = 2. Floyd Warshall's will process sp(i,j,0) for all other pairs considering only vertex 0 as the intermediate vertex but there is only one more change: sp(3,1,0) from ∞ down to 3.
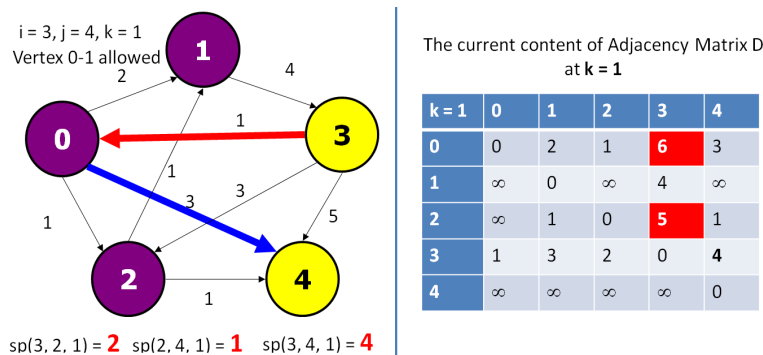
Figure 4.22: Floyd Warshall's Explanation 3

When we allow $k = 1$, i.e. vertex 0 and 1 can now be used as intermediate vertices, then it happens that there is no change to sp(3,2,1), sp(2,4,1), nor to sp(3,4,1). However, two other values change: sp(0,3,1) and sp(2,3,1) as shown in Figure 4.22 but these two values will not affect the final computation of the shortest path between vertex 3 and 4.
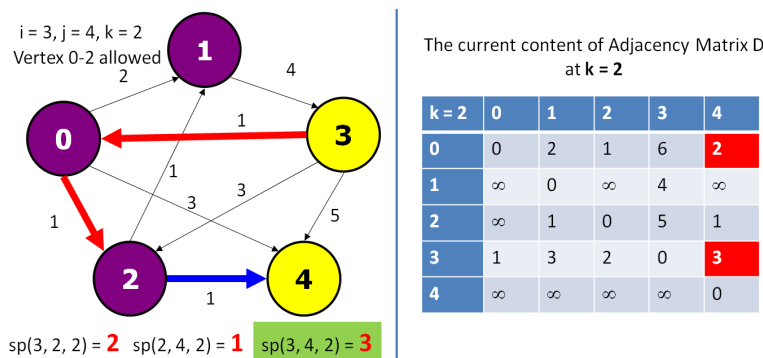
Figure 4.23: Floyd Warshall's Explanation 4

When we allow $k = 2$, i.e. vertex 0, 1, and 2 now can be used as the intermediate vertices, then sp(3,4,2) is reduced again as sp(3,4,2) = sp(3,2,2)+sp(2,4,2) = 2+1 = 3 as shown in Figure 4.23. Floyd Warshall's repeats this process for $k = 3$ and finally $k = 4$ but sp(3,4,4) remains at 3 and this is the final answer.

Formally, we define Floyd Warshall's DP recurrences as follow. Let $D_{i,j}^k$ be the shortest distance from i to j with only [0..k] as intermediate vertices. Then, Floyd Warshall's base case and recurrence are as follow:

$D_{i,j}^{-1} = weight(i,j)$. This is the base case when we do not use any intermediate vertices.
$D_{i,j}^k = min(D_{i,j}^{k-1}, D_{i,k}^{k-1} + D_{k,j}^{k-1}) = min(\text{not using vertex } k, \text{using vertex } k)$, for $k \geq 0$.

This DP formulation must be filled layer by layer (by increasing k). To fill out an entry in the table k, we make use of the entries in the table k-1. For example, to calculate $D_{3,4}^2$, (row 3, column 4, in table $k = 2$, index start from 0), we look at the minimum of $D_{3,4}^1$ or the sum of $D_{3,2}^1 + D_{2,4}^1$ (see Table 4.3). The naïve implementation is to use a 3-dimensional matrix D[k][i][j] of size $O(V^3)$. However, since to compute layer k we only need to know the values from layer k-1, we can drop dimension k and compute D[i][j] 'on-the-fly' (the space saving trick discussed in Section 3.5.1). Thus, Floyd Warshall's algorithm just need $O(V^2)$ space although it still runs in $O(V^3)$.

157

Table 4.3: Floyd Warshall's DP Table

4.5.3 Other Applications

The main purpose of Floyd Warshall's is to solve the APSP problem. However, Floyd Warshall's is frequently used in other problems too, as long as the input graph is small. Here we list down several problem variants that are also solvable with Floyd Warshall's.

Solving the SSSP Problem on a Small Weighted Graph

If we have the All-Pairs Shortest Paths (APSP) information, we also know the Single-Source Shortest Paths (SSSP) information from any possible source. If the given weighted graph is small $V \leq 400$, it may be beneficial, in terms of coding time, to use the four-liner Floyd Warshall's code rather than the longer Dijkstra's algorithm.

Printing the Shortest Paths

A common issue encountered by programmers who use the four-liner Floyd Warshall's without understanding how it works is when they are asked to print the shortest paths too. In BFS/Dijkstra's/Bellman Ford's algorithms, we just need to remember the shortest paths spanning tree by using a 1D `vi p` to store the parent information for each vertex. In Floyd Warshall's, we need to store a 2D parent matrix. The modified code is shown below.

```
// inside int main()
  // let p be a 2D parent matrix, where p[i][j] is the last vertex before j
  // on a shortest path from i to j, i.e. i -> ... -> p[i][j] -> j
  for (int i = 0; i < V; i++)
    for (int j = 0; j < V; j++)
      p[i][j] = i;                          // initialize the parent matrix
  for (int k = 0; k < V; k++)
    for (int i = 0; i < V; i++)
      for (int j = 0; j < V; j++) // this time, we need to use if statement
        if (AdjMat[i][k] + AdjMat[k][j] < AdjMat[i][j]) {
          AdjMat[i][j] = AdjMat[i][k] + AdjMat[k][j];
          p[i][j] = p[k][j];                // update the parent matrix
        }
//----------------------------------------------------------------------
// when we need to print the shortest paths, we can call the method below:
void printPath(int i, int j) {
  if (i != j) printPath(i, p[i][j]);
  printf(" %d", j);
}
```

Transitive Closure (Warshall's Algorithm)

Stephen *Warshall* [70] developed an algorithm for the Transitive Closure problem: Given a graph, determine if vertex i is connected to j, directly *or indirectly*. This variant uses logical bitwise operators which is (much) faster than arithmetic operators. Initially, `AdjMat[i][j]` contains 1 (`true`) if vertex i is *directly* connected to vertex j, 0 (`false`) otherwise. After running $O(V^3)$ Warshall's algorithm below, we can check if any two vertices i and j are directly or indirectly connected by checking `AdjMat[i][j]`.

```
for (int k = 0; k < V; k++)
  for (int i = 0; i < V; i++)
    for (int j = 0; j < V; j++)
      AdjMat[i][j] |= (AdjMat[i][k] & AdjMat[k][j]);
```

Minimax and Maximin (Revisited)

We have seen the minimax (and maximin) path problem earlier in Section 4.3.4. The solution using Floyd Warshall's is shown below. First, initialize `AdjMat[i][j]` to be the weight of edge `(i,j)`. This is the default minimax cost for two vertices that are directly connected. For pair `i-j` without any direct edge, set `AdjMat[i][j] = INF`. Then, we try all possible intermediate vertex k. The minimax cost `AdjMat[i][j]` is the minimum of either (itself) or (the maximum between `AdjMat[i][k]` or `AdjMat[k][j]`). However, this approach can only be used if the input graph is small enough ($V \leq 400$).

```
for (int k = 0; k < V; k++)
  for (int i = 0; i < V; i++)
    for (int j = 0; j < V; j++)
      AdjMat[i][j] = min(AdjMat[i][j], max(AdjMat[i][k], AdjMat[k][j]));
```

Finding the (Cheapest/Negative) Cycle

In Section 4.4.4, we have seen how Bellman Ford's terminates after $O(VE)$ steps regardless of the type of input graph (as it relax all E edges at most V-1 times) and how Bellman Ford's can be used to check if the given graph has negative cycle. Floyd Warshall's also terminates after $O(V^3)$ steps regardless of the type of input graph. This feature allows Floyd Warshall's to be used to detect whether the (small) graph has a cycle, a negative cycle, and even finding the cheapest (non-negative) cycle among all possible cycles (the girth of the graph).

To do this, we initially set the *main diagonal* of the Adjacency Matrix to have a very large value, i.e. `AdjMat[i][i] = INF` (1B). Then, we run the $O(V^3)$ Floyd Warshall's algorithm. Now, we check the value of `AdjMat[i][i]`, which now means the shortest cyclic path weight starting from vertex `i` that goes through up to V-1 other intermediate vertices and returns back to `i`. If `AdjMat[i][i]` is no longer INF for any i $\in$ [0..V-1], then we have a cycle. The smallest non-negative `AdjMat[i][i]` $\forall$i $\in$ [0..V-1] is the *cheapest* cycle. If `AdjMat[i][i] < 0` for any i $\in$ [0..V-1], then we have a *negative* cycle because if we take this cyclic path one more time, we will get an even shorter 'shortest' path.

Finding the Diameter of a Graph

The diameter of a graph is defined as the maximum shortest path distance between any pair of vertices of that graph. To find the diameter of a graph, we first find the shortest path

between each pair of vertices (i.e. the APSP problem). The maximum distance found is the diameter of the graph. UVa 1056 - Degrees of Separation, which is an ICPC World Finals problem in 2006, is precisely this problem. To solve this problem, we can first run an $O(V^3)$ Floyd Warshall's to compute the required APSP information. Then, we can figure out what is the diameter of the the graph by finding the maximum value in the `AdjMat` in $O(V^2)$. However, we can only do this for a small graph with $V \leq 400$.

Finding the SCCs of a Directed Graph

In Section 4.2.1, we have learned how the $O(V + E)$ Tarjan's algorithm can be used to identify the SCCs of a directed graph. However, the code is a bit long. If the input graph is small (e.g. UVa 247 - Calling Circles, UVa 1229 - Sub-dictionary, UVa 10731 - Test), we can also identify the SCCs of the graph in $O(V^3)$ using Warshall's transitive closure algorithm and then use the following check: To find all members of an SCC that contains vertex `i`, check all other vertices `j` ∈ `[0..V-1]`. If `AdjMat[i][j] && AdjMat[j][i]` is true, then vertex `i` and `j` belong to the same SCC.

Exercise 4.5.3.1: How to find the transitive closure of a graph with $V \leq 1000, E \leq 100000$? Suppose that there are only Q $(1 \leq 100 \leq Q)$ transitive closure queries for this problem in form of this question: Is vertex u connected to vertex v, directly or indirectly? What if the input graph is *directed*? Does this directed property simplify the problem?

Exercise 4.5.3.2*: Solve the *maximin* path problem using Floyd Warshall's!

Exercise 4.5.3.3: Arbitrage is the trading of one currency for another with the hopes of taking advantage of small differences in conversion rates among several currencies in order to achieve a profit. For example (UVa 436 - Arbitrage II): If 1.0 United States dollar (USD) buys 0.5 British pounds (GBP), 1.0 GBP buys 10.0 French francs (FRF[12]), and 1.0 FRF buys 0.21 USD, then an arbitrage trader can start with 1.0 USD and buy $1.0 \times 0.5 \times 10.0 \times 0.21 = 1.05$ USD thus earning a profit of 5 percent. This problem is actually a problem of finding a *profitable cycle*. It is akin to the problem of finding cycle with Floyd Warshall's shown in this section. Solve the arbitrage problem using Floyd Warshall's!

Remarks About Shortest Paths in Programming Contests

All three algorithms discussed in the past two sections: Dijkstra's, Bellman Ford's, and Floyd Warshall's are used to solve the *general case* of shortest paths (SSSP or APSP) problems on weighted graphs. Out of these three, the $O(VE)$ Bellman Ford's is rarely used in programming contests due to its high time complexity. It is only useful if the problem author gives a 'reasonable size' graph with negative cycle. For general cases, (our modified) $O((V+E) \log V)$ Dijkstra's implementation variant is the best solution for the SSSP problem for 'reasonable size' weighted graph without negative cycle. However, when the given graph is small ($V \leq 400$)—which happens many times, it is clear from this section that the $O(V^3)$ Floyd Warshall's is the best way to go.

One possible reason on why Floyd Warshall's algorithm is quite popular in programming contests is because sometimes the problem author includes shortest paths as the *sub-problem* of the main, (much) more complex, problem. To make the problem still doable during contest time, the problem author purposely sets the input size to be small so that the shortest paths

---

[12]At the moment (2013), France actually uses Euro as its currency.

sub-problem is solvable with the four liner Floyd Warshall's (e.g. UVa 10171, 10793, 11463). A non competitive programmer will take longer route to deal with this sub-problem.

According to our experience, many shortest paths problems are not on weighted graphs that require Dijkstra's or Floyd Warshall's algorithms. If you look at the programming exercises listed in Section 4.4 (and later in Section 8.2), you will see that many of them are on unweighted graphs that are solvable with BFS (see Section 4.4.2).

We also observe that today's trend related to shortest paths problem involves careful *graph modeling* (UVa 10067, 10801, 11367, 11492, 12160). Therefore, to do well in programming contests, make sure that you have this soft skill: The ability to spot the graph in the problem statement. We have shown several examples of such graph modeling skill in this chapter which we hope you are able to appreciate and eventually make it yours.

In Section 4.7.1, we will revisit some shortest paths problems on Directed Acyclic Graph (DAG). This important variant is solvable with generic Dynamic Programming (DP) technique that have been discussed in Section 3.5. We will also present another way of viewing DP technique as 'algorithm on DAG' in that section.

We present an SSSP/APSP algorithm decision table within the context of programming contest in Table 4.4 to help the readers in deciding which algorithm to choose depending on various graph criteria. The terminologies used are as follows: 'Best' → the most suitable algorithm; 'Ok' → a correct algorithm but not the best; 'Bad' → a (very) slow algorithm; 'WA' → an incorrect algorithm; and 'Overkill' → a correct algorithm but unnecessary.

Graph Criteria	BFS $O(V + E)$	Dijkstra's $O((V+E)\log V)$	Bellman Ford's $O(VE)$	Floyd Warshall's $O(V^3)$
Max Size	$V, E \leq 10M$	$V, E \leq 300K$	$VE \leq 10M$	$V \leq 400$
Unweighted	Best	Ok	Bad	Bad in general
Weighted	WA	Best	Ok	Bad in general
Negative weight	WA	Our variant Ok	Ok	Bad in general
Negative cycle	Cannot detect	Cannot detect	Can detect	Can detect
Small graph	WA if weighted	Overkill	Overkill	Best

Table 4.4: SSSP/APSP Algorithm Decision Table

Programming Exercises for Floyd Warshall's algorithm:

- Floyd Warshall's Standard Application (for APSP or SSSP on small graph)

    1. UVa 00341 - Non-Stop Travel (graph is small)
    2. UVa 00423 - MPI Maelstrom (graph is small)
    3. UVa 00567 - Risk (simple SSSP solvable with BFS, but graph is small, so can be solved easier with Floyd Warshall's)
    4. **UVa 00821 - Page Hopping *** (LA 5221, World Finals Orlando00, one of the 'easiest' ICPC World Finals problem)
    5. UVa 01233 - USHER (LA 4109, Singapore07, Floyd Warshall's can be used to find the minimum cost cycle in the graph; the maximum input graph size is $p \leq 500$ but still doable in UVa online judge)
    6. UVa 01247 - Interstar Transport (LA 4524, Hsinchu09, APSP, Floyd Warshall's, modified a bit to prefer shortest path with less intermediate vertices)
    7. **UVa 10171 - Meeting Prof. Miguel *** (easy with APSP information)
    8. *UVa 10354 - Avoiding Your Boss* (find boss's shortest paths, remove edges involved in boss's shortest paths, re-run shortest paths from home to market)

9. *UVa 10525 - New to Bangladesh?* (use two adjacency matrices: time and length; use modified Floyd Warshall's)
10. UVa 10724 - Road Construction (adding one edge only changes 'a few things')
11. UVa 10793 - The Orc Attack (Floyd Warshall's simplifies this problem)
12. UVa 10803 - Thunder Mountain (graph is small)
13. UVa 10947 - Bear with me, again.. (graph is small)
14. UVa 11015 - 05-32 Rendezvous (graph is small)
15. **UVa 11463 - Commandos *** (solution is easy with APSP information)
16. *UVa 12319 - Edgetown's Traffic Jams* (Floyd Warshall's 2x and compare)

- Variants

1. **UVa 00104 - Arbitrage *** (small arbitrage problem solvable with FW)
2. UVa 00125 - Numbering Paths (modified Floyd Warshall's)
3. UVa 00186 - Trip Routing (graph is small, print path)
4. *UVa 00274 - Cat and Mouse* (variant of transitive closure problem)
5. UVa 00436 - Arbitrage (II) (another arbitrage problem)
6. **UVa 00334 - Identifying Concurrent ... *** (transitive closure++)
7. UVa 00869 - Airline Comparison (run Warshall's 2x, compare AdjMatrices)
8. *UVa 00925 - No more prerequisites ...* (transitive closure++)
9. **UVa 01056 - Degrees of Separation *** (LA 3569, World Finals SanAntonio06, diameter of a small graph)
10. *UVa 01198 - Geodetic Set Problem* (LA 2818, Kaohsiung03, trans closure++)
11. UVa 11047 - The Scrooge Co Problem (print path; special case: if origin = destination, print twice)

Profile of Algorithm Inventors

Robert W Floyd (1936-2001) was an eminent American computer scientist. Floyd's contributions include the design of **Floyd's algorithm** [19], which efficiently finds all shortest paths in a graph. Floyd worked closely with Donald Ervin Knuth, in particular as the major reviewer for Knuth's seminal book The Art of Computer Programming, and is the person most cited in that work.

Stephen Warshall (1935-2006) was a computer scientist who invented the **transitive closure algorithm**, now known as **Warshall's algorithm** [70]. This algorithm was later named as Floyd Warshall's as Floyd independently invented essentially similar algorithm.

Jack R. Edmonds (born 1934) is a mathematician. He and Richard Karp invented the **Edmonds Karp's algorithm** for computing the Max Flow in a flow network in $O(VE^2)$ [14]. He also invented an algorithm for MST on directed graphs (Arborescence problem). This algorithm was proposed independently first by Chu and Liu (1965) and then by Edmonds (1967)—thus called the **Chu Liu/Edmonds's algorithm** [6]. However, his most important contribution is probably the Edmonds's **matching/blossom shrinking algorithm**—one of the most cited Computer Science papers [13].

Richard Manning Karp (born 1935) is a computer scientist. He has made many important discoveries in computer science in the area of combinatorial algorithms. In 1971, he and Edmonds published the **Edmonds Karp's algorithm** for solving the Max Flow problem [14]. In 1973, he and John Hopcroft published the **Hopcroft Karp's algorithm**, still the fastest known method for finding Maximum Cardinality Bipartite Matching [28].

4.6 Network Flow

4.6.1 Overview and Motivation

Motivating problem: Imagine a connected, (integer) weighted, and directed graph[13] as a pipe network where the edges are the pipes and the vertices are the splitting points. Each edge has a weight equals to the capacity of the pipe. There are also two special vertices: source s and sink t. What is the maximum flow (rate) from source s to sink t in this graph (imagine water flowing in the pipe network, we want to know the maximum volume of water over time that can pass through this pipe network)? This problem is called the Maximum Flow problem (often abbreviated as just Max Flow), one of the problems in the family of problems involving flow in networks. See an illustration of Max Flow in Figure 4.24.
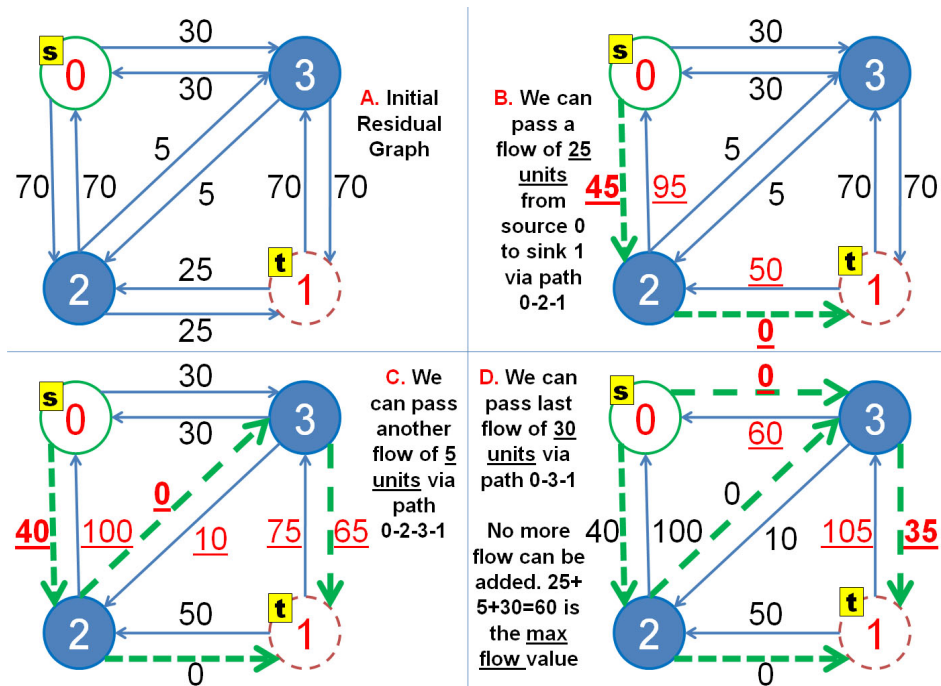
Figure 4.24: Max Flow Illustration (UVa 820 [47] - ICPC World Finals 2000 Problem E)

4.6.2 Ford Fulkerson's Method

One solution for Max Flow is the Ford Fulkerson's method—invented by the same Lester Randolph *Ford*, Jr who invented the Bellman Ford's algorithm and Delbert Ray *Fulkerson*.

```
setup directed residual graph with edge capacity = original graph weights
mf = 0              // this is an iterative algorithm, mf stands for max_flow
while (there exists an augmenting path p from s to t) {
  // p is a path from s to t that pass through +ve edges in residual graph
  augment/send flow f along the path p (s -> ... -> i -> j -> ... t)
    1. find f, the minimum edge weight along the path p
    2. decrease capacity of forward edges (e.g. i -> j) along path p by f
    3. increase capacity of backward edges (e.g. j -> i) along path p by f
  mf += f           // we can send a flow of size f from s to t, increase mf
}
output mf                                   // this is the max flow value
```

[13]A weighted edge in an undirected graph can be transformed to two directed edges with the same weight.

Ford Fulkerson's method is an iterative algorithm that repeatedly finds augmenting path p: A path from source s to sink t that passes through positive weighted edges in the residual[14] graph. After finding an augmenting path p that has f as the minimum edge weight along the path p (the bottleneck edge in this path), Ford Fulkerson's method will do two important steps: Decreasing/increasing the capacity of forward $(i \to j)$/backward $(j \to i)$ edges along path p by f, respectively. Ford Fulkerson's method will repeat this process until there is no more possible augmenting path from source s to sink t anymore which implies that the total flow so far is the maximum flow. Now see Figure 4.24 again with this understanding.

The reason for decreasing the capacity of forward edge is obvious. By sending a flow through augmenting path p, we will decrease the remaining (residual) capacities of the (forward) edges used in p. The reason for increasing the capacity of backward edges may not be that obvious, but this step is important for the correctness of Ford Fulkerson's method. By increasing the capacity of a backward edge $(j \to i)$, Ford Fulkerson's method allows *future iteration (flow)* to cancel (part of) the capacity used by a forward edge $(i \to j)$ that was incorrectly used by some earlier flow(s).

There are several ways to find an augmenting s-t path in the pseudo code above, each with different behavior. In this section, we highlight two ways: via DFS or via BFS.

Ford Fulkerson's method implemented using DFS may run in $O(|f^*|E)$ where $|f^*|$ is the Max Flow `mf` value. This is because we can have a graph like in Figure 4.25. Then, we may encounter a situation where two augmenting paths: $s \to a \to b \to t$ and $s \to b \to a \to t$ only decrease the (forward[15]) edge capacities along the path by 1. In the worst case, this is repeated $|f^*|$ times (it is 200 times in Figure 4.25). Because DFS runs in $O(E)$ in a flow graph[16], the overall time complexity is $O(|f^*|E)$. We do not want this unpredictability in programming contests as the problem author can choose to give a (very) large $|f^*|$ value.
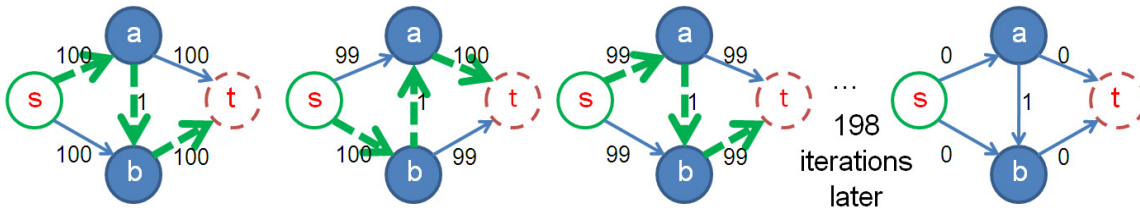
Figure 4.25: Ford Fulkerson's Method Implemented with DFS Can Be Slow

4.6.3 Edmonds Karp's Algorithm

A better implementation of the Ford Fulkerson's method is to use BFS for finding the shortest path in terms of number of layers/hops between s and t. This algorithm was discovered by Jack *Edmonds* and Richard Manning *Karp*, thus named as Edmonds Karp's algorithm [14]. It runs in $O(VE^2)$ as it can be proven that after $O(VE)$ BFS iterations, all augmenting paths will already be exhausted. Interested readers can browse references like [14, 7] to study more about this proof. As BFS runs in $O(E)$ in a flow graph, the overall time complexity is $O(VE^2)$. Edmonds Karp's only needs two s-t paths in Figure 4.25: s→a→t (2 hops, send

---

[14]We use the name 'residual graph' because initially the weight of each edge `res[i][j]` is the same as the original capacity of edge `(i, j)` in the original graph. If this edge `(i, j)` is used by an augmenting path and a flow pass through this edge with weight `f` $\leq$ `res[i][j]` (a flow cannot exceed this capacity), then the remaining (or residual) capacity of edge `(i, j)` will be `res[i][j] - f`.

[15]Note that after sending flow $s \to a \to b \to t$, the forward edge $a \to b$ is replaced by the backward edge $b \to a$, and so on. If this is not so, then the max flow value is just $1 + 99 + 99 = 199$ instead of 200 (wrong).

[16]The number of edges in a flow graph must be $E \geq V - 1$ to ensure $\exists \geq 1$ s-t flow. This implies that both DFS and BFS—using Adjacency List—run in $O(E)$ instead of $O(V + E)$.

100 unit flow) and s→b→t (2 hops, send another 100). That is, it does not get trapped to
send flow via the longer paths (3 hops): s→a→b→t (or s→b→a→t).

Coding the Edmonds Karp's algorithm for the first time can be a challenge for new
programmers. In this section, we provide our simplest Edmonds Karp's code that uses *only*
Adjacency Matrix named as `res` with size $O(V^2)$ to store the residual capacity of each edge.
This version—which runs in $O(VE)$ BFS iterations $\times$ $O(V^2)$ per BFS due to Adjacency
Matrix $= O(V^3E)$—is fast enough to solve *some* (small-size) Max Flow problems.

```
int res[MAX_V][MAX_V], mf, f, s, t;                              // global variables
vi p;                                         // p stores the BFS spanning tree from s

void augment(int v, int minEdge) {  // traverse BFS spanning tree from s->t
  if (v == s) { f = minEdge; return; }  // record minEdge in a global var f
  else if (p[v] != -1) { augment(p[v], min(minEdge, res[p[v]][v]));
                         res[p[v]][v] -= f; res[v][p[v]] += f; } }

// inside int main(): set up 'res', 's', and 't' with appropriate values
  mf = 0;                                               // mf stands for max_flow
  while (1) {           // O(VE^2) (actually O(V^3 E) Edmonds Karp's algorithm
    f = 0;
    // run BFS, compare with the original BFS shown in Section 4.2.2
    vi dist(MAX_V, INF); dist[s] = 0; queue<int> q; q.push(s);
    p.assign(MAX_V, -1);       // record the BFS spanning tree, from s to t!
    while (!q.empty()) {
      int u = q.front(); q.pop();
      if (u == t) break; // immediately stop BFS if we already reach sink t
      for (int v = 0; v < MAX_V; v++)                     // note: this part is slow
        if (res[u][v] > 0 && dist[v] == INF)
          dist[v] = dist[u] + 1, q.push(v), p[v] = u;       // 3 lines in 1!
    }
    augment(t, INF);   // find the min edge weight 'f' in this path, if any
    if (f == 0) break; // we cannot send any more flow ('f' = 0), terminate
    mf += f;                  // we can still send a flow, increase the max flow!
  }
  printf("%d\n", mf);                               // this is the max flow value
```

Visualization: `www.comp.nus.edu.sg/~stevenha/visualization/maxflow.html`

Source code: `ch4_08_edmonds_karp.cpp/java`

Exercise 4.6.3.1: Before continuing, answer the following question in Figure 4.26!
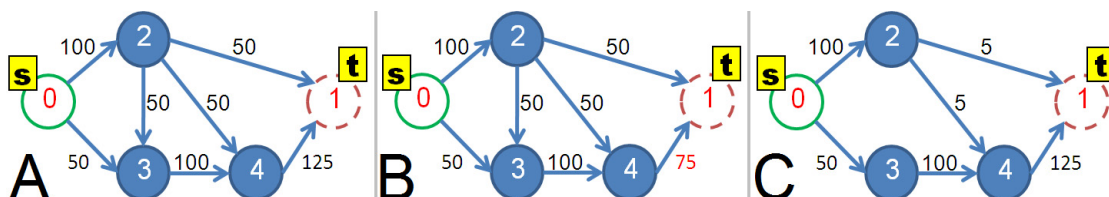
Figure 4.26: What are the Max Flow value of these three residual graphs?

Exercise 4.6.3.2: The main weakness of the simple code shown in this section is that enumerating the neighbors of a vertex takes $O(V)$ instead of $O(k)$ (where k is the number of neighbors of that vertex). The other (but not significant) weakness is that we also do not need `vi dist` as `bitset` (to flag whether a vertex has been visited or not) is sufficient. Modify the Edmonds Karp's code above so that it achieves its $O(VE^2)$ time complexity!

Exercise 4.6.3.3*: An even better implementation of the Edmonds Karp's algorithm is to avoid using the $O(V^2)$ Adjacency Matrix to store the residual capacity of each edge. A better way is to store both the original capacity and the actual flow (not just the residual) of each edge as an $O(V + E)$ modified Adjacency + Edge List. This way, we have three information for each edge: The original capacity of the edge, the flow currently in the edge, and we can derive the residual of an edge from the original capacity minus the flow of that edge. Now, modify the implementation again! How to handle the backward flow efficiently?

4.6.4 Flow Graph Modeling - Part 1

With the given Edmonds Karp's code above, solving a (basic/standard) Network Flow problem, especially Max Flow, is now simpler. It is now a matter of:

1. Recognizing that the problem is indeed a Network Flow problem
   (this will get better after you solve more Network Flow problems).

2. Constructing the appropriate flow graph (i.e. if using our code shown earlier: Initiate the residual matrix `res` and set the appropriate values for 's' and 't').

3. Running the Edmonds Karp's code on this flow graph.

In this subsection, we show an example of *modeling* the flow (residual) graph of UVa 259 - Software Allocation[17]. The abridged version of this problem is as follows: You are given up to 26 applications/apps (labeled 'A' to 'Z'), up to 10 computers (numbered from 0 to 9), the number of persons who brought in each application that day (one digit positive integer, or [1..9]), the list of computers that a particular application can run, and the fact that each computer can only run one application that day. Your task is to determine whether an allocation (that is, a *matching*) of applications to computers can be done, and if so, generates a possible allocation. If no, simply print an exclamation mark '!'.

One (bipartite) flow graph formulation is shown in Figure 4.27. We index the vertices from [0..37] as there are 26 + 10 + 2 special vertices = 38 vertices. The source s is given index 0, the 26 possible apps are given indices from [1..26], the 10 possible computers are given indices from [27..36], and finally the sink t is given index 37.
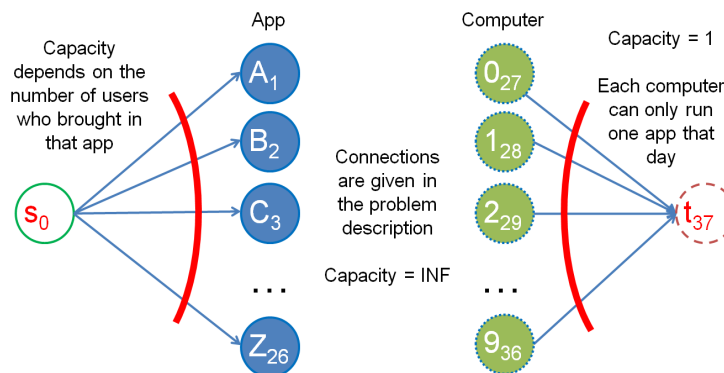
Figure 4.27: Residual Graph of UVa 259 [47]

---

[17]Actually this problem has small input size (we only have 26 + 10 = 36 vertices plus 2 more: source and sink) which make this problem still solvable with recursive backtracking (see Section 3.2). The name of this problem is 'assignment problem' or (special) bipartite matching with capacity.

Then, we link apps to computers as mentioned in the problem description. We link source s to all apps and link all computers to sink t. All edges in this flow graph are *directed* edges. The problem says that there can be *more than one* (say, X) users bringing in a particular app A in a given day. Thus, we set the edge weight (capacity) from source s to a particular app A to X. The problem also says that each computer can only be used once. Thus, we set the edge weight from each computer B to sink t to 1. The edge weight between apps to computers is set to ∞. With this arrangement, if there is a flow from an app A to a computer B and finally to sink t, that flow corresponds to *one matching* between that particular app A and computer B.

Once we have this flow graph, we can pass it to our Edmonds Karp's implementation shown earlier to obtain the Max Flow `mf`. If `mf` is equal to the number of applications brought in that day, then we have a solution, i.e. if we have X users bringing in app A, then X different paths (i.e. matchings) from A to sink t must be found by the Edmonds Karp's algorithm (and similarly for the other apps).

The actual app $\rightarrow$ computer assignments can be found by simply checking the backward edges from computers (vertices 27 - 36) to apps (vertices 1 - 26). A backward edge (computer $\rightarrow$ app) in the residual matrix `res` will contain a value $+1$ if the corresponding forward edge (app $\rightarrow$ computer) is selected in the paths that contribute to the Max Flow `mf`. This is also the reason why we start the flow graph with *directed* edges from apps to computers only.

Exercise 4.6.4.1: Why do we use ∞ for the edge weights (capacities) of directed edges from apps to computers? Can we use capacity 1 instead of ∞?

Exercise 4.6.4.2*: Is this kind of assignment problem (bipartite matching with capacity) can be solved with standard Max Cardinality Bipartite Matching (MCBM) algorithm shown later in Section 4.7.4? If it is possible, determine which one is the better solution?

4.6.5 Other Applications

There are several other interesting applications/variants of the problems involving flow in a network. We discuss three examples here while some others are deferred until Section 4.7.4 (Bipartite Graph), Section 9.13, Section 9.22, and Section 9.23. Note that some tricks shown here may also be applicable to other graph problems.

Minimum Cut

Let's define an s-t cut $C = (S\text{-component}, T\text{-component})$ as a partition of $V \in G$ such that source $s \in S$-component and sink $t \in T$-component. Let's also define a *cut-set* of C to be the set $\{(u, v) \in E \mid u \in S\text{-component}, v \in T\text{-component}\}$ such that if all edges in the cut-set of C are removed, the Max Flow from s to t is 0 (i.e. s and t are disconnected). The cost of an s-t cut C is defined by the sum of the capacities of the edges in the cut-set of C. The Minimum Cut problem, often abbreviated as just Min Cut, is to minimize the amount of capacity of an s-t cut. This problem is more general than finding bridges (see Section 4.2.1), i.e. in this case we can cut *more* than just one edge and we want to do so in the least cost way. As with bridges, Min Cut has applications in 'sabotaging' networks, e.g. One pure Min Cut problem is UVa 10480 - Sabotage.

The solution is simple: The by-product of computing Max Flow is Min Cut! Let's see Figure 4.24.D again. After Max Flow algorithm stops, we run graph traversal (DFS/BFS) from source s again. All reachable vertices from source s using positive weighted edges in the residual graph belong to the S-component (i.e. vertex 0 and 2). All other unreachable

vertices belong to the T-component (i.e. vertex 1 and 3). All edges connecting the S-component to the T-component belong to the cut-set of C (edge 0-3 (capacity 30/flow 30/residual 0), 2-3 (5/5/0) and 2-1 (25/25/0) in this case). The Min Cut value is 30+5+25 = 60 = Max Flow value `mf`. This is the minimum over all possible s-t cuts value.

Multi-source/Multi-sink

Sometimes, we can have more than one source and/or more than one sink. However, this variant is no harder than the original Network Flow problem with a single source and a single sink. Create a super source ss and a super sink st. Connect ss with all s with infinite capacity and also connect all t with st with infinite capacity, then run Edmonds Karp's as per normal. Note that we have seen this variant in **Exercise 4.4.2.1**.

Vertex Capacities
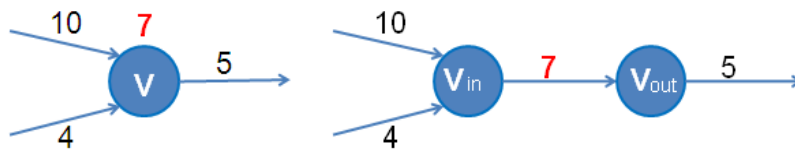
Figure 4.28: Vertex Splitting Technique

We can also have a Network Flow variant where the capacities are not just defined along the edges but *also on the vertices*. To solve this variant, we can use *vertex splitting* technique which (unfortunately) *doubles* the number of vertices in the flow graph. A weighted graph with a vertex weight can be converted into a more familiar one *without* vertex weight by splitting each weighted vertex v to v_{in} and v_{out}, reassigning its incoming/outgoing edges to v_{in}/v_{out}, respectively and finally putting the original vertex v's weight as the weight of edge $v_{in} \to v_{out}$. See Figure 4.28 for illustration. Now with all weights defined on edges, we can run Edmonds Karp's as per normal.

4.6.6 Flow Graph Modeling - Part 2

The hardest part of dealing with Network Flow problem is the modeling of the flow graph (assuming that we already have a good pre-written Max Flow code). In Section 4.6.4, we have seen one example modeling to deal with the assignment problem (or bipartite matching with capacity). Here, we present another (harder) flow graph modeling for UVa 11380 - Down Went The Titanic. Our advice before you continue reading: Please do not just memorize the solution but also try to understand the key steps to derive the required flow graph.
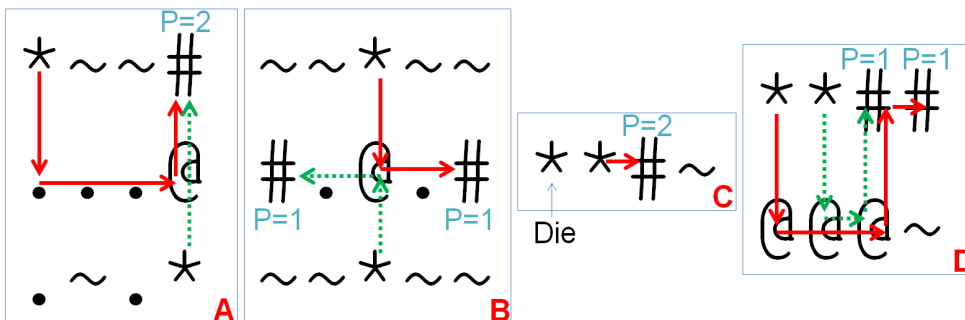
Figure 4.29: Some Test Cases of UVa 11380

168

In Figure 4.29, we have four small test cases of UVa 11380. You are given a small 2D grid containing these five characters as shown in Table 4.5. You want to put as many '*' (people) as possible to the (various) safe place(s): the '#' (large wood). The solid and dotted arrows in Figure 4.29 denotes the answer.

Symbol	Meaning	# Usage	Capacity
*	People staying on floating ice	1	1
~	Freezing water	0	0
.	Floating ice	1	1
@	Large iceberg	∞	1
#	Large wood	∞	P

Table 4.5: Characters Used in UVa 11380

To model the flow graph, we use the following thinking steps. In Figure 4.30.A, we first connect non '~' cells together with large capacity (1000 is enough for this problem). This describes the possible movements in the grid. In Figure 4.30.B, we set vertex capacities of '*' and '.' cells to 1 to indicate that they can only be used *once*. Then, we set vertex capacities of '@' and '#' to a large value (1000) to indicate that they can be used *several times*. In Figure 4.30.C, we create a source vertex s and sink vertex t. Source s is linked to all '*' cells in the grid with capacity 1 to indicate that there is one person to be saved. All '#' cells in the grid are connected to sink t with capacity P to indicate that the large wood can be used P times. Now, the required answer—the number of survivor(s)—equals to the max flow value between source s and sink t of this flow graph. As the flow graph uses vertex capacities, we need to use the *vertex splitting* technique discussed earlier.
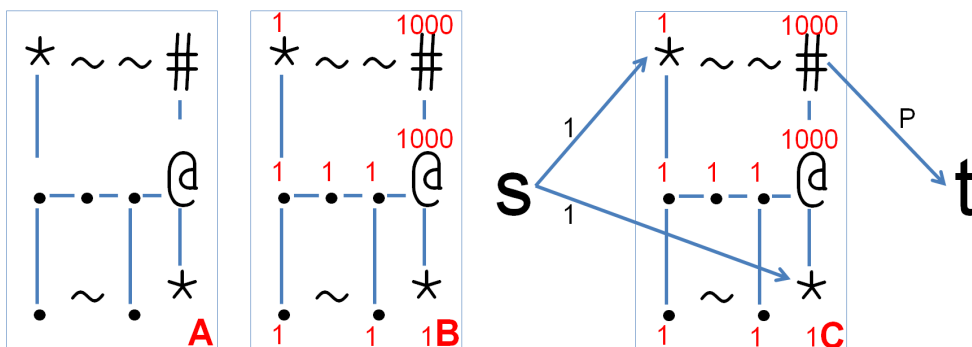
Figure 4.30: Flow Graph Modeling

Exercise 4.6.6.1*: Does $O(VE^2)$ Edmonds Karp's fast enough to compute the max flow value on the largest possible flow graph of UVa 11380: 30×30 grid and $P = 10$? Why?

Remarks About Network Flow in Programming Contests

As of 2013, when a Network (usually Max) Flow problem appears in a programming contest, it is *usually* one of the 'decider' problems. In ICPC, many interesting graph problems are written in such a way that they do not look like a Network Flow in a glance. The hardest part for the contestant is to realize that the underlying problem is indeed a Network Flow problem and able to model the flow graph correctly. This is the key skill that has to be mastered via practice.

To avoid wasting precious contest time coding the relatively long Max Flow library code, we suggest that in an ICPC team, one team member devotes significant effort in preparing a good Max Flow code (perhaps Dinic's algorithm implementation, see Section 9.7) and attempts various Network Flow problems available in many online judges to increase his/her familiarity towards Network Flow problems and its variants. In the list of programming exercises in this section, we have some simple Max Flow, bipartite matching with capacity (the assignment problem), Min Cut, and network flow problems involving vertex capacities. Try to solve as many programming exercises as possible.

In Section 4.7.4, we will see the classic Max Cardinality Bipartite Matching (MCBM) problem and see that this problem is also solvable with Max Flow. Later in Chapter 9, we will see some harder problems related to Network Flow, e.g. a faster Max Flow algorithm (Section 9.7), the Independent and Edge-Disjoint Paths problems (Section 9.13), the Max *Weighted* Independent Set on Bipartite Graph problem (Section 9.22), and the Min Cost (Max) Flow problem (Section 9.23).

In IOI, Network Flow (and its variants) is currently outside the 2009 syllabus [20]. So, IOI contestants can choose to skip this section. However, we believe that it is a good idea for IOI contestants to learn these more advanced material 'ahead of time' to improve your skills with graph problems.

Programming Exercises related to Network Flow:

- Standard Max Flow Problem (Edmonds Karp's)
    1. **UVa 00259 - Software Allocation *** (discussed in this section)
    2. **UVa 00820 - Internet Bandwidth *** (LA 5220, World Finals Orlando00, basic max flow, discussed in this section)
    3. UVa 10092 - The Problem with the ... (assignment problem, matching with capacity, similar with UVa 259)
    4. UVa 10511 - Councilling (matching, max flow, print the assignment)
    5. *UVa 10779 - Collectors Problem* (max flow modeling is not straightforward; the main idea is to build a flow graph such that each augmenting path corresponds to a series of exchange of duplicate stickers, starting with Bob giving away one of his duplicates, and ending with him receiving a new sticker; repeat until this is no longer possible)
    6. UVa 11045 - My T-Shirt Suits Me (assignment problem; but actually the input constraint is actually small enough for recursive backtracking)
    7. **UVa 11167 - Monkeys in the Emei ... *** (max flow modeling; there are lots of edges in the flow graph; therefore, it is better to compress the capacity-1 edges whenever possible; use $O(V^2E)$ Dinic's max flow algorithm so that the high number of edges does not penalize the performance of your solution)
    8. UVa 11418 - Clever Naming Patterns (two layers of matching, it may be easier to use max flow solution)
- Variants
    1. UVa 10330 - Power Transmission (max flow with vertex capacities)
    2. UVa 10480 - Sabotage (straightforward min cut problem)
    3. **UVa 11380 - Down Went The Titanic *** (discussed in this section)
    4. **UVa 11506 - Angry Programmer *** (min cut with vertex capacities)
    5. *UVa 12125 - March of the Penguins *** (max flow modeling with vertex capacities; another interesting problem, similar level with UVa 11380)

4.7   Special Graphs

Some basic graph problems have simpler/faster polynomial algorithms if the given graph is *special*. Based on our experience, we have identified the following special graphs that commonly appear in programming contests: **Directed Acyclic Graph (DAG)**, **Tree**, **Eulerian Graph**, and **Bipartite Graph**. Problem authors may force the contestants to use specialized algorithms for these special graphs by giving a large input size to judge a correct algorithm for general graph as Time Limit Exceeded (TLE) (see a survey by [21]). In this section, we discuss some popular graph problems on these special graphs (see Figure 4.31)—many of which have been discussed earlier on general graphs. Note that at the time of writing, bipartite graph (Section 4.7.4) is still excluded in the IOI syllabus [20].
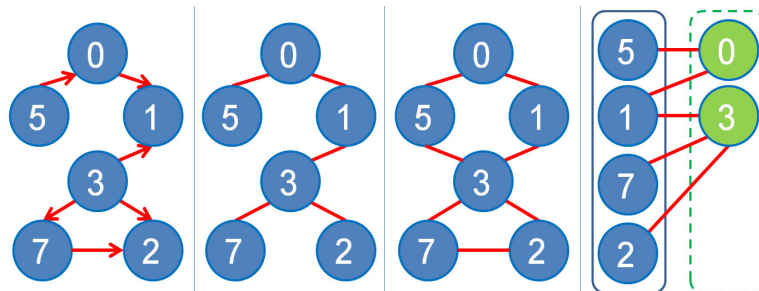
Figure 4.31: Special Graphs (L-to-R): DAG, Tree, Eulerian, Bipartite Graph

4.7.1   Directed Acyclic Graph

A Directed Acyclic Graph (DAG) is a special graph with the following characteristics: Directed and has no cycle. DAG guarantees the absence of cycle *by definition*. This makes problems that can be modeled as a DAG very suitable to be solved with Dynamic Programming (DP) techniques (see Section 3.5). After all, a DP recurrence must be *acyclic*. We can view DP states as vertices in an implicit DAG and the acyclic transitions between DP states as directed edges of that implicit DAG. Topological sort of this DAG (see Section 4.2.1) allows each overlapping subproblem (subgraph of the DAG) to be processed just once.

(Single-Source) Shortest/Longest Paths on DAG

The Single-Source Shortest Paths (SSSP) problem becomes much simpler if the given graph is a DAG. This is because a DAG has at least one topological order! We can use an $O(V+E)$ topological sort algorithm in Section 4.2.1 to find one such topological order, then relax the outgoing edges of these vertices according to this order. The topological order will ensure that if we have a vertex b that has an incoming edge from a vertex a, then vertex b is relaxed *after* vertex a has obtained correct shortest distance value. This way, the shortest distance values propagation is correct with just one $O(V + E)$ linear pass! This is also the essence of Dynamic Programming principle to avoid recomputation of overlapping subproblem covered earlier in Section 3.5. When we compute bottom-up DP, we essentially fill the DP table using the topological order of the underlying implicit DAG of DP recurrences.

The (Single-Source)[18] *Longest Paths* problem is a problem of finding the longest (simple[19]) paths from a starting vertex s to other vertices. The decision version of this problem

---

[18]Actually this can be multi-sources, as we can start from any vertex with 0 incoming degree.

[19]On general graph with positive weighted edges, the longest path problem is ill-defined as one can take a positive cycle and use that cycle to create an infinitely long path. This is the same issue as the negative cycle in shortest path problem. That is why for general graph, we use the term: 'longest *simple* path problem'. All paths in DAG are simple by definition so we can just use the term 'longest path problem'.

is NP-complete on a general graph[20]. However the problem is again easy if the graph has no cycle, which is true in a DAG. The solution for the Longest Paths on DAG[21] is just a minor tweak from the DP solution for the SSSP on DAG shown above. One trick is to multiply all edge weights by -1 and run the same SSSP solution as above. Finally, negate the resulting values to get the actual results.

The Longest Paths on DAG has applications in project scheduling, e.g. UVa 452 - Project Scheduling about Project Evaluation and Review Technique (PERT). We can model sub projects dependency as a DAG and the time needed to complete a sub project as *vertex weight*. The shortest possible time to finish the entire project is determined by the longest path in this DAG (a.k.a. the *critical* path) that starts from any vertex (sub project) with 0 incoming degree. See Figure 4.32 for an example with 6 sub projects, their estimated completion time units, and their dependencies. The longest path $0 \to 1 \to 2 \to 4 \to 5$ with 16 time units determines the shortest possible time to finish the whole project. In order to achieve this, all sub projects along the longest (critical) path must be on time.
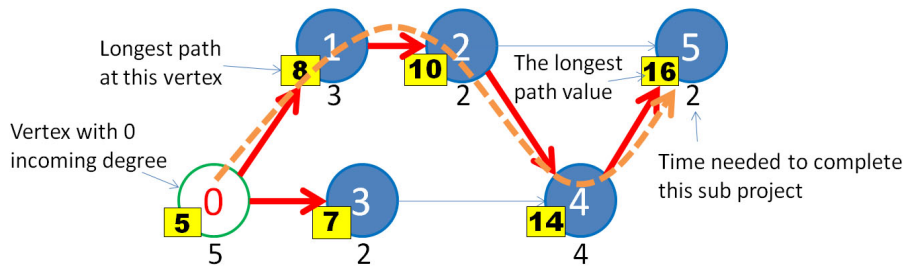
Figure 4.32: The Longest Path on this DAG

Counting Paths in DAG

Motivating problem (UVa 988 - Many paths, one destination): In life, one has many paths to choose, leading to many different lives. Enumerate how many different lives one can live, given a specific set of choices at each point in time. One is given a list of events, and a number of choices that can be selected, for each event. The objective is to count how many ways to go from the event that started it all (birth, index 0) to an event where one has no further choices (that is, death, index n).
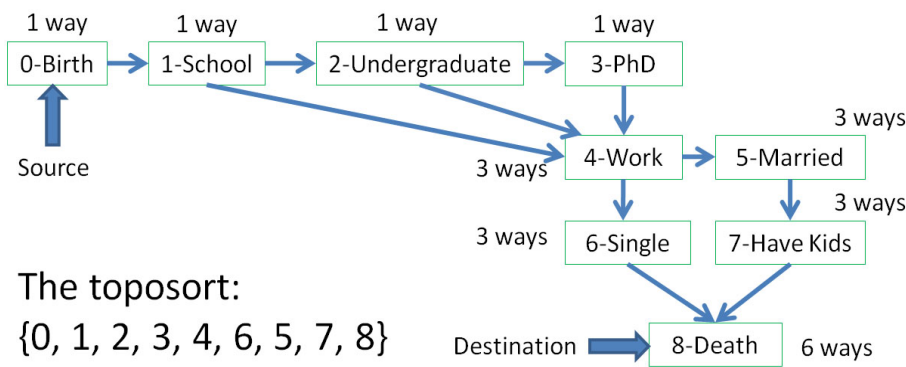
Figure 4.33: Example of Counting Paths in DAG - Bottom-Up

Clearly the underlying graph of the problem above is a DAG as one can move forward in time, but cannot go backward. The number of such paths can be found easily by computing one (any) topological order in $O(V + E)$ (in this problem, vertex 0/birth will always be the

---

[20]The decision version of this problem asks if the general graph has a simple path of total weight $\geq k$.
[21]The LIS problem in Section 3.5.2 can also be modeled as finding the Longest Paths on implicit DAG.

first in the topological order and the vertex n/death will always be the last in the topological order). We start by setting `num_paths[0] = 1`. Then, we process the remaining vertices one by one according to the topological order. When processing a vertex `u`, we update each neighbor `v` of `u` by setting `num_paths[v] += num_paths[u]`. After such $O(V + E)$ steps, we will know the number of paths in `num_paths[n]`. Figure 4.33 shows an example with 9 events and eventually 6 different possible life scenarios.

Bottom-Up versus Top-Down Implementations

Before we continue, we want to remark that all three solutions for shortest/longest/counting paths on/in DAG above are Bottom-Up DP solutions. We start from known base case(s) (the source vertex/vertices) and then we use topological order of the DAG to propagate the correct information to neighboring vertices without ever needing to backtrack.

We have seen in Section 3.5 that DP can also be written in Top-Down fashion. Using UVa 988 as an illustration, we can also write the DP solution as follows: Let `numPaths(i)` be the number of paths starting from vertex i to destination n. We can write the solution using this Complete Search recurrence relations:

1. `numPaths(n) = 1 // at destination n, there is only one possible path`
2. `numPaths(i) = ` $\sum_j$ ` numPaths(j),` $\forall$`j adjacent to i`

To avoid recomputations, we memoize the number of paths for each vertex i. There are $O(V)$ distinct vertices (states) and each vertex is only processed once. There are $O(E)$ edges and each edge is also visited at most once. Therefore the time complexity of this Top-Down approach is also $O(V + E)$, same as the Bottom-Up approach shown earlier. Figure 4.34 shows the similar DAG but the values are computed from destination to source (follow the dotted back arrows). Compare this Figure 4.34 with the previous Figure 4.33 where the values are computed from source to destination.
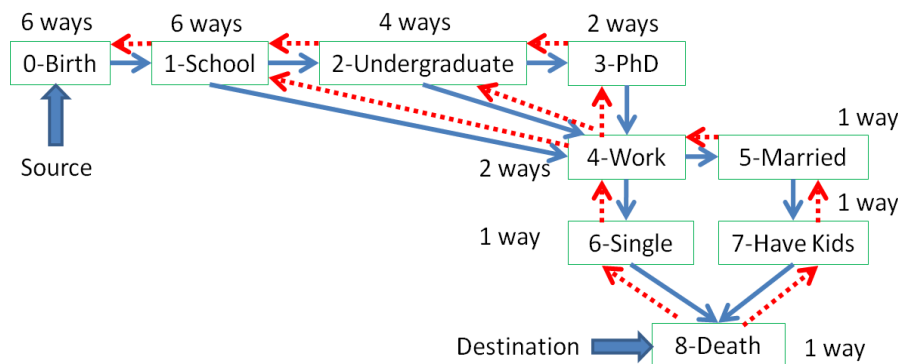
Figure 4.34: Example of Counting Paths in DAG - Top-Down

Converting General Graph to DAG

Sometimes, the given graph in the problem statement is not an *explicit DAG*. However, after further understanding, the given graph can be modeled as a DAG if we add one (or more) parameter(s). Once you have the DAG, the next step is to apply Dynamic Programming technique (either Top-Down or Bottom-Up). We illustrate this concept with two examples.

1. SPOJ 0101: Fishmonger

Abridged problem statement: Given the number of cities $3 \leq n \leq 50$, available time $1 \leq t \leq 1000$, and two $n \times n$ matrices (one gives travel times and another gives tolls between two cities), choose a route from the port city (vertex 0) in such a way that the fishmonger has to

pay as little tolls as possible to arrive at the market city (vertex $n-1$) within a certain time t. The fishmonger does *not* have to visit all cities. Output two information: The total tolls that is actually used and the actual traveling time. See Figure 4.35—left, for the original input graph of this problem.

Notice that there are *two* potentially conflicting requirements in this problem. The first requirement is to *minimize* tolls along the route. The second requirement is to *ensure* that the fishmonger arrive in the market city within allocated time, which may cause him to pay higher tolls in some part along the path. The second requirement is a *hard* constraint for this problem. That is, we must satisfy it, otherwise we do not have a solution.
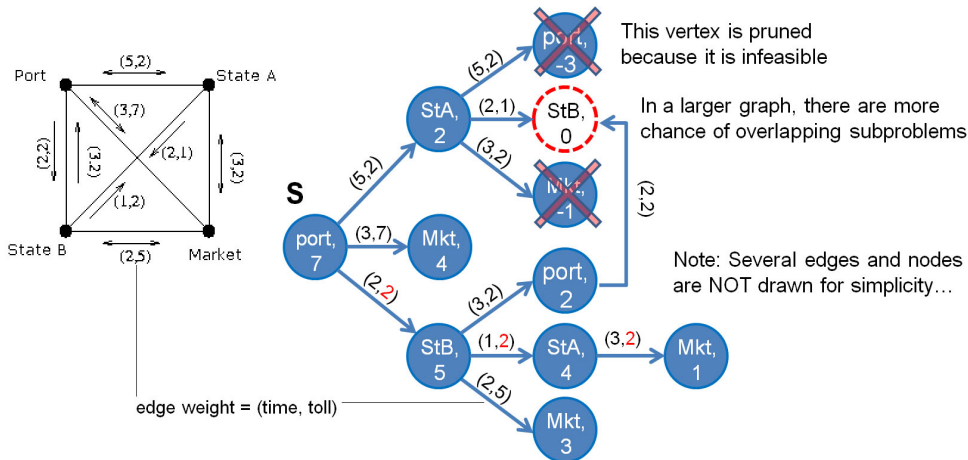
Figure 4.35: The Given General Graph (left) is Converted to DAG

Greedy SSSP algorithm like Dijkstra's (see Section 4.4.3)—on its pure form—does not work for this problem. Picking a path with the shortest travel time to help the fishmonger to arrive at market city $n-1$ using time $\leq t$ may not lead to the smallest possible tolls. Picking path with the cheapest tolls may not ensure that the fishmonger arrives at market city $n-1$ using time $\leq t$. These two requirements are not independent!

However, if we attach a parameter: `t_left` (time left) to each vertex, then the given graph turns into a DAG as shown in Figure 4.35, right. We start with a vertex `(port, t)` in the DAG. Every time the fishmonger moves from a current city `cur` to another city `X`, we move to a modified vertex `(X, t - travelTime[cur][X])` in the DAG via edge with weight `toll[cur][X]`. As time is a diminishing resource, we will never encounter a cyclic situation. We can then use this (Top-Down) DP recurrence: `go(cur, t_left)` to find the shortest path (in terms of total tolls paid) on this DAG. The answer can be found by calling `go(0, t)`. The C++ code of `go(cur, t_left)` is shown below:

```
ii go(int cur, int t_left) {        // returns a pair (tollpaid, timeneeded)
  if (t_left < 0) return ii(INF, INF);              // invalid state, prune
  if (cur == n - 1) return ii(0, 0); // at market, tollpaid=0, timeneeded=0
  if (memo[cur][t_left] != ii(-1, -1)) return memo[cur][t_left];
  ii ans = ii(INF, INF);
  for (int X = 0; X < n; X++) if (cur != X) {         // go to another city
    ii nextCity = go(X, t_left - travelTime[cur][X]);    // recursive step
    if (nextCity.first + toll[cur][X] < ans.first) {   // pick the min cost
      ans.first  = nextCity.first  + toll[cur][X];
      ans.second = nextCity.second + travelTime[cur][X];
  } }
  return memo[cur][t_left] = ans; }        // store the answer to memo table
```

Notice that by using Top-Down DP, we do not have to explicitly build the DAG and compute the required topological order. The recursion will do these steps for us. There are only $O(nt)$ distinct states (notice that the memo table is a pair object). Each state can be computed in $O(n)$. The overall time complexity is thus $O(n^2 t)$—do-able.

2. Minimum Vertex Cover (on a Tree)

The tree data structure is also an acyclic data structure. But unlike DAG, there are no overlapping subtrees in a tree. Thus, there is no point of using Dynamic Programming (DP) technique on a standard tree. However, similar with the Fishmonger example above, some trees in programming contest problems turn into DAGs if we attach one (or more) parameter(s) to each vertex of the tree. Then, the solution is usually to run DP on the resulting DAG. Such problems are (inappropriately[22]) named as the 'DP on Tree' problems in competitive programming terminology.
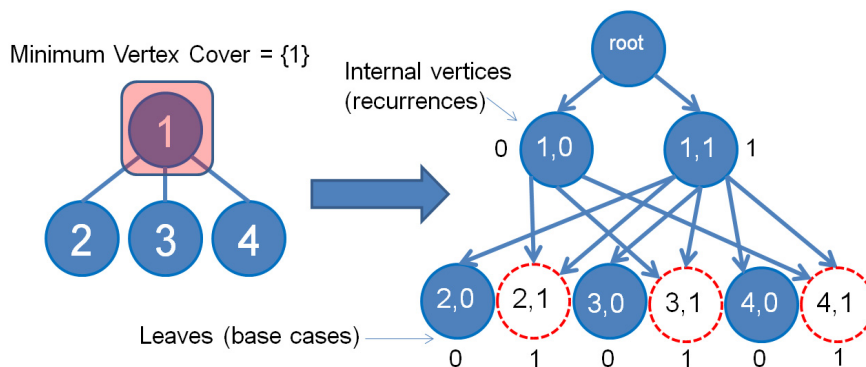
Figure 4.36: The Given General Graph/Tree (left) is Converted to DAG

An example of this DP on Tree problem is the problem of finding the Minimum Vertex Cover (MVC) on a Tree. In this problem, we have to select the smallest possible set of vertices $C \in V$ such that each edge of the tree is incident to at least one vertex of the set C. For the sample tree shown in Figure 4.36—left, the solution is to take vertex 1 only, because all edges 1-2, 1-3, 1-4 are all incident to vertex 1.

Now, there are only two possibilities for each vertex. Either it is taken, or it is not. By attaching this 'taken or not taken' status to each vertex, we convert the tree into a DAG (see Figure 4.36—right). Each vertex now has (vertex number, boolean flag taken/not). The implicit edges are determined with the following rules: 1). If the current vertex is not taken, then we have to take all its children to have a valid solution. 2). If the current vertex is taken, then we take the best between taking or not taking its children. We can now write this top down DP recurrences: `MVC(v, flag)`. The answer can be found by calling `min(MVC(root, false), MVC(root, true))`. Notice the presence of overlapping subproblems (dotted circles) in the DAG. However, as there are only $2 \times V$ states and each vertex has at most two incoming edges, this DP solution runs in $O(V)$.

```
int MVC(int v, int flag) {                            // Minimum Vertex Cover
  int ans = 0;
  if (memo[v][flag] != -1) return memo[v][flag];              // top down DP
  else if (leaf[v])      // leaf[v] is true if v is a leaf, false otherwise
    ans = flag;                                          // 1/0 = taken/not
```

---

[22]We have mentioned that there is no point of using DP on a Tree. But the term 'DP on Tree' that actually refers to 'DP on implicit DAG' is already a well-known term in competitive programming community.

```
  else if (flag == 0) {      // if v is not taken, we must take its children
    ans = 0;       // Note: 'Children' is an Adjacency List that contains the
    // directed version of the tree (parent points to its children; but the
    // children does not point to parents)
    for (int j = 0; j < (int)Children[v].size(); j++)
      ans += MVC(Children[v][j], 1);
  }
  else if (flag == 1) {              // if v is taken, take the minimum between
    ans = 1;                                // taking or not taking its children
    for (int j = 0; j < (int)Children[v].size(); j++)
      ans += min(MVC(Children.[v][j], 1), MVC(Children[v][j], 0));
  }
  return memo[v][flag] = ans;
}
```

Section 3.5—Revisited

Here, we want to re-highlight to the readers the strong linkage between DP techniques shown in Section 3.5 and algorithms on DAG. Notice that all programming exercises about shortest/longest/counting paths on/in DAG (or on general graph that is converted to DAG by some graph modeling/transformation) can also be classified under DP category. Often when we have a problem with DP solution that 'minimizes this', 'maximizes that', or 'counts something', that DP solution actually computes the shortest, the longest, or count the number of paths on/in the (usually implicit) DP recurrence DAG of that problem, respectively.

We now invite the readers to revisit some DP problems that we have seen earlier in Section 3.5 with this likely new viewpoint (viewing DP as algorithms on DAG is not commonly found in other Computer Science textbooks). As a start, we revisit the classic Coin Change problem. Figure 4.37 shows the same test case used in Section 3.5.2. There are $n = 2$ coin denominations: $\{1, 5\}$. The target amount is $V = 10$. We can model each vertex as the current value. Each vertex v has $n = 2$ unweighted edges that goes to vertex $v - 1$ and $v - 5$ in this test case, unless if it causes the index to go negative. Notice that the graph is a DAG and some states (highlighted with dotted circles) are overlapping (have more than one incoming edges). Now, we can solve this problem by finding the *shortest path* on this DAG from source $V = 10$ to target $V = 0$. The easiest topological order is to process the vertices in reverse sorted order, i.e. $\{10, 9, 8, \ldots, 1, 0\}$ is a valid topological order. We can definitely use the $O(V + E)$ shortest paths on DAG solution. However, since the graph is unweighted, we can also use the $O(V + E)$ BFS to solve this problem (using Dijkstra's is also possible but overkill). The path: $10 \to 5 \to 0$ is the shortest with total weight $= 2$ (or 2 coins needed). Note: For this test case, a greedy solution for coin change will also pick the same path: $10 \to 5 \to 0$.
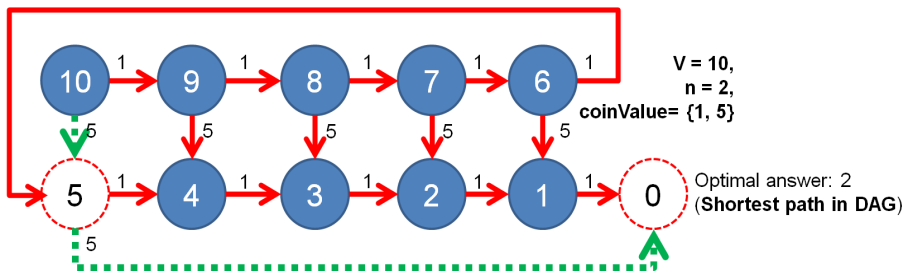
Figure 4.37: Coin Change as Shortest Paths on DAG

Next, let's revisit the classic 0-1 Knapsack Problem. This time we use the following test case: $n = 5, V = \{4, 2, 10, 1, 2\}, W = \{12, 1, 4, 1, 2\}, S = 15$. We can model each vertex as a pair of values (`id, remW`). Each vertex has at least one edge (`id, remW`) to (`id+1, remW`) that corresponds to not taking a certain item `id`. Some vertices have edge (`id, remW`) to (`id+1, remW-W[id]`) if `W[id]` $\leq$ `remW` that corresponds to taking a certain item `id`. Figure 4.38 shows some parts of the computation DAG of the standard 0-1 Knapsack Problem using the test case above. Notice that some states can be visited with more than one path (an overlapping subproblem is highlighted with a dotted circle). Now, we can solve this problem by finding the *longest path* on this DAG from the source (`0, 15`) to target (`5, any`). The answer is the following path: (`0, 15`) $\rightarrow$ (`1, 15`) $\rightarrow$ (`2, 14`) $\rightarrow$ (`3, 10`) $\rightarrow$ (`4, 9`) $\rightarrow$ (`5, 7`) with weight $0 + 2 + 10 + 1 + 2 = 15$.
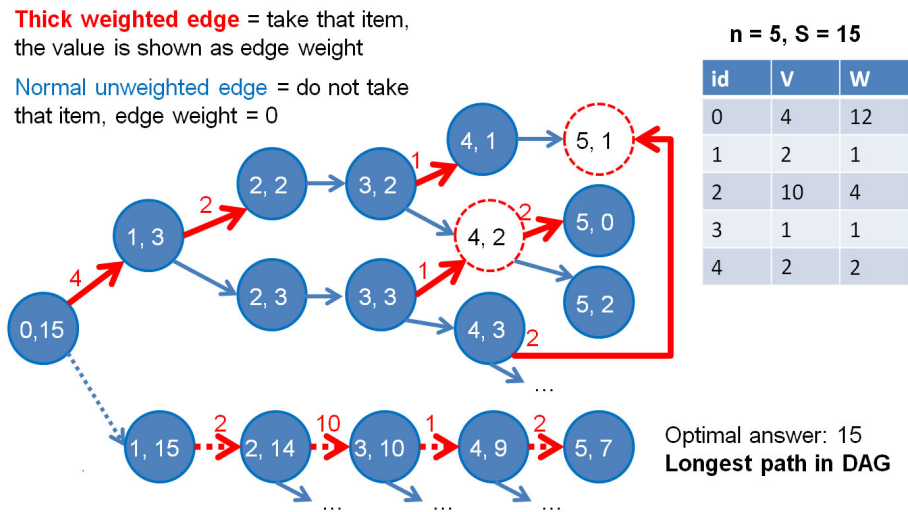
Figure 4.38: 0-1 Knapsack as Longest Paths on DAG

Let's see one more example: The solution for UVa 10943 - How do you add? discussed in Section 3.5.3. If we draw the DAG of this test case: $n = 3, K = 4$, then we have a DAG as shown in Figure 4.39. There are overlapping subproblems highlighted with dotted circles. If we count the number of paths in this DAG, we will indeed find the answer $= 20$ paths.
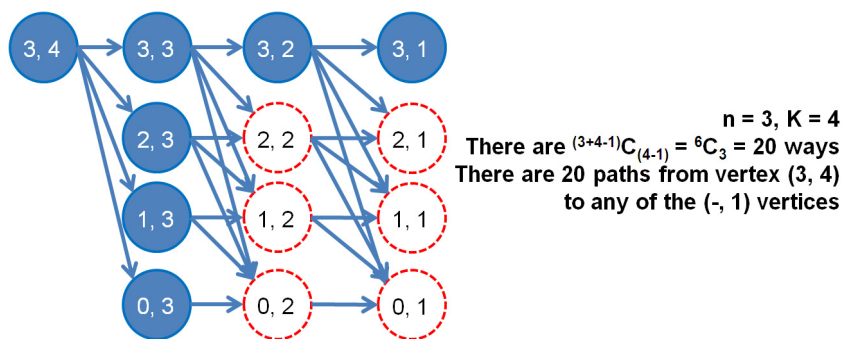
Figure 4.39: UVa 10943 as Counting Paths in DAG

Exercise 4.7.1.1*: Draw the DAG for some test cases of the other classical DP problems not mentioned above: Traveling Salesman Problem (TSP) $\approx$ shortest paths on the implicit DAG, Longest Increasing Subsequence (LIS) $\approx$ longest paths of the implicit DAG, Counting Change variant (the one about counting the number of possible ways to get value V cents using a list of denominations of N coins) $\approx$ counting paths in DAG, etc.

4.7.2 Tree

Tree is a special graph with the following characteristics: It has $E = V$-1 (any $O(V + E)$ algorithm on tree is $O(V)$), it has no cycle, it is connected, and there exists one unique path for any pair of vertices.

Tree Traversal

In Section 4.2.1 and 4.2.2, we have seen $O(V + E)$ DFS and BFS algorithms for traversing a general graph. If the given graph is a *rooted binary tree*, there are *simpler* tree traversal algorithms like pre-order, in-order, and post-order traversal (note: level-order traversal is essentially BFS). There is no major time speedup as these tree traversal algorithms also run in $O(V)$, but the code are simpler. Their pseudo-code are shown below:

```
pre-order(v)                in-order(v)                 post-order(v)
  visit(v);                   in-order(left(v));          post-order(left(v));
  pre-order(left(v));         visit(v);                   post-order(right(v));
  pre-order(right(v));        in-order(right(v));         visit(v);
```

Finding Articulation Points and Bridges in Tree

In Section 4.2.1, we have seen $O(V + E)$ Tarjan's DFS algorithm for finding articulation points and bridges of a graph. However, if the given graph is a tree, the problem becomes simpler: All edges on a tree are bridges and all internal vertices (degree > 1) are articulation points. This is still $O(V)$ as we have to scan the tree to count the number of internal vertices, but the code is *simpler*.

Single-Source Shortest Paths on Weighted Tree

In Sections 4.4.3 and 4.4.4, we have seen two general purpose algorithms ($O((V + E) \log V)$ Dijkstra's and $O(VE)$ Bellman-Ford's) for solving the SSSP problem on a weighted graph. But if the given graph is a weighted tree, the SSSP problem becomes *simpler*: Any $O(V)$ graph traversal algorithm, i.e. BFS or DFS, can be used to solve this problem. There is only one unique path between any two vertices in a tree, so we simply traverse the tree to find the unique path connecting the two vertices. The shortest path weight between these two vertices is basically the sum of edge weights of this unique path (e.g. from vertex 5 to vertex 3 in Figure 4.40.A, the unique path is 5->0->1->3 with weight 4+2+9 = 15).

All-Pairs Shortest Paths on Weighted Tree

In Section 4.5, we have seen a general purpose algorithm ($O(V^3)$ Floyd Warshall's) for solving the APSP problem on a weighted graph. However, if the given graph is a weighted tree, the APSP problem becomes *simpler*: Repeat the SSSP on weighted tree V times, setting each vertex as the source vertex one by one. The overall time complexity is $O(V^2)$.

Diameter of Weighted Tree

For general graph, we need $O(V^3)$ Floyd Warshall's algorithm discussed in Section 4.5 plus another $O(V^2)$ all-pairs check to compute the diameter. However, if the given graph is a weighted tree, the problem becomes *simpler*. We only need two $O(V)$ traversals: Do DFS/BFS from *any* vertex s to find the furthest vertex x (e.g. from vertex s=1 to vertex x=2 in Figure 4.40.B1), then do DFS/BFS one more time from vertex x to get the true

furthest vertex y from x. The length of the unique path along x to y is the diameter of that tree (e.g. path x=2->3->1->0->y=5 with length 20 in Figure 4.40.B2).
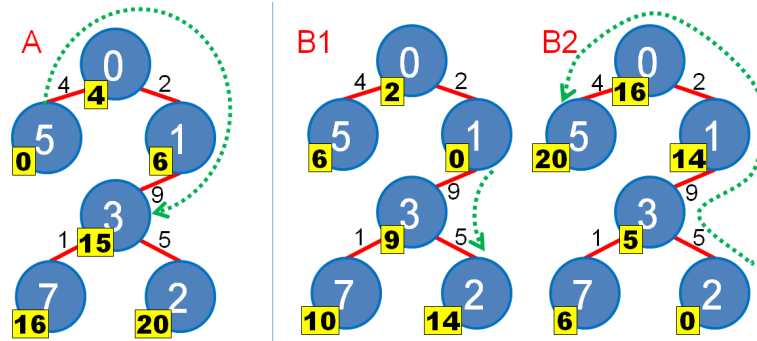
Figure 4.40: A: SSSP (Part of APSP); B1-B2: Diameter of Tree

Exercise 4.7.2.1*: Given the inorder and preorder traversal of a rooted Binary Search Tree (BST) T containing n vertices, write a recursive pseudo-code to output the postorder traversal of that BST. What is the time complexity of your best algorithm?

Exercise 4.7.2.2*: There is an even faster solution than $O(V^2)$ for the All-Pairs Shortest Paths problem on Weighted Tree. It uses LCA. How?

4.7.3 Eulerian Graph

An *Euler path* is defined as a path in a graph which visits *each edge* of the graph *exactly once*. Similarly, an *Euler tour/cycle* is an Euler path which starts and ends on the same vertex. A graph which has either an Euler path or an Euler tour is called an Eulerian graph[23].

This type of graph is first studied by Leonhard *Euler* while solving the Seven Bridges of Königsberg problem in 1736. Euler's finding 'started' the field of graph theory!
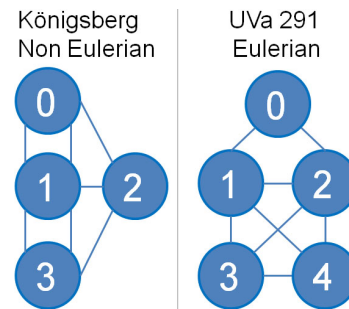
Figure 4.41: Eulerian

Eulerian Graph Check

To check whether a connected undirected graph has an Euler tour is simple. We just need to check if all its vertices have even degrees. It is similar for the Euler path, i.e. an undirected graph has an Euler path if all except two vertices have even degrees. This Euler path will start from one of these odd degree vertices and end in the other[24]. Such degree check can be done in $O(V + E)$, usually done simultaneously when reading the input graph. You can try this check on the two graphs in Figure 4.41.

Printing Euler Tour

While checking whether a graph is Eulerian is easy, finding the actual Euler tour/path requires more work. The code below produces the desired Euler tour when given an unweighted Eulerian graph stored in an Adjacency List where the second attribute in edge information pair is a Boolean 1 (this edge can still be used) or 0 (this edge can no longer be used).

---

[23]Compare this property with the *Hamiltonian path/cycle* in TSP (see Section 3.5.2).

[24]Euler path on *directed graph* is also possible: Graph must be connected, has equal in/outdegree vertices, at most one vertex with indegree - outdegree = 1, and at most one vertex with outdegree - indegree = 1.

```
list<int> cyc;                 // we need list for fast insertion in the middle

void EulerTour(list<int>::iterator i, int u) {
  for (int j = 0; j < (int)AdjList[u].size(); j++) {
    ii v = AdjList[u][j];
    if (v.second) {            // if this edge can still be used/not removed
      v.second = 0;     // make the weight of this edge to be 0 ('removed')
      for (int k = 0; k < (int)AdjList[v.first].size(); k++) {
        ii uu = AdjList[v.first][k];        // remove bi-directional edge
        if (uu.first == u && uu.second) {
          uu.second = 0;
          break;
      } }
      EulerTour(cyc.insert(i, u), v.first);
} } }

// inside int main()
  cyc.clear();
  EulerTour(cyc.begin(), A);    // cyc contains an Euler tour starting at A
  for (list<int>::iterator it = cyc.begin(); it != cyc.end(); it++)
    printf("%d\n", *it);                             // the Euler tour
```

4.7.4   Bipartite Graph

Bipartite Graph is a special graph with the following characteristics: The set of vertices V can be partitioned into two disjoint sets V_1 and V_2 and all edges in $(u, v) \in E$ has the property that $u \in V_1$ and $v \in V_2$. This makes a Bipartite Graph free from odd-length cycles (see **Exercise 4.2.6.3**). Note that Tree is also a Bipartite Graph!

Max Cardinality Bipartite Matching (MCBM) and Its Max Flow Solution

Motivating problem (from TopCoder Open 2009 Qualifying 1 [31]): Given a list of numbers N, return a list of all the elements in N that can be paired with $N[0]$ successfully as part of a *complete prime pairing*, sorted in ascending order. Complete prime pairing means that each element a in N is paired to a unique other element b in N such that $a + b$ is prime.

For example: Given a list of numbers $N = \{1, 4, 7, 10, 11, 12\}$, the answer is $\{4, 10\}$. This is because pairing $N[0] = 1$ with 4 results in a prime pair and the other four items can also form two prime pairs ($7 + 10 = 17$ and $11 + 12 = 23$). Similar situation by pairing $N[0] = 1$ with 10, i.e. $1 + 10 = 11$ is a prime pair and we also have two other prime pairs ($4 + 7 = 11$ and $11 + 12 = 23$). We cannot pair $N[0] = 1$ with any other item in N. For example, if we pair $N[0] = 1$ with 12, we have a prime pair but there will be no way to pair the remaining 4 numbers to form 2 more prime pairs.

Constraints: List N contains an even number of elements ([2..50]). Each element of N will be between [1..1000]. Each element of N will be distinct.

Although this problem involves prime numbers, it is not a pure math problem as the elements of N are not more than 1K—there are not too many primes below 1000 (only 168 primes). The issue is that we cannot do Complete Search pairings as there are $_{50}C_2$ possibilities for the first pair, $_{48}C_2$ for the second pair, ..., until $_2C_2$ for the last pair. DP with bitmask technique (Section 8.3.1) is also not usable because 2^{50} is too big.

The key to solve this problem is to realize that this pairing (matching) is done on *bipartite graph*! To get a prime number, we need to sum 1 odd + 1 even, because 1 odd + 1 odd (or 1 even + 1 even) produces an even number (which is not prime). Thus we can split odd/even numbers to `set1`/`set2` and add edge `i → j` if `set1[i] + set2[j]` is prime.
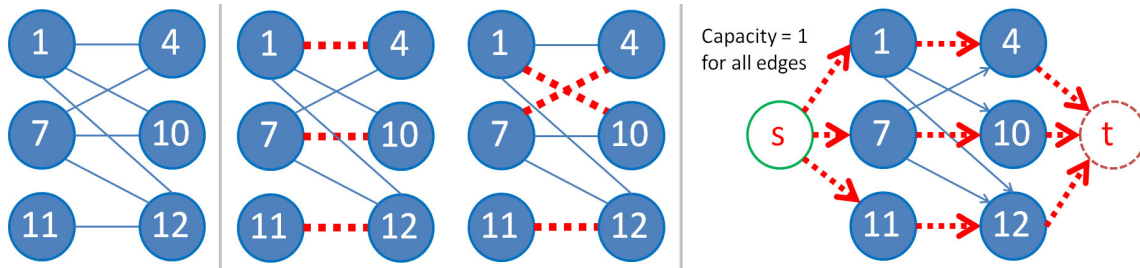
Figure 4.42: Bipartite Matching problem can be reduced to a Max Flow problem

After we build this bipartite graph, the solution is trivial: If the size of `set1` and `set2` are different, a complete pairing is not possible. Otherwise, if the size of both sets are $n/2$, try to match `set1[0]` with `set2[k]` for $k = $ `[0..n/2-1]` and do Max Cardinality Bipartite Matching (MCBM) for the rest (MCBM is one of the most common applications involving Bipartite Graph). If we obtain $n/2 - 1$ more matchings, add `set2[k]` to the answer. For this test case, the answer is {4, 10} (see Figure 4.42, middle).

MCBM problem can be reduced to the Max Flow problem by assigning a dummy source vertex s connected to all vertices in `set1` and all vertices in `set2` are connected to a dummy sink vertex t. The edges are directed ($s → u$, $u → v$, $v → t$ where $u \in$ `set1` and $v \in$ `set2`). By setting the capacities of all edges in this flow graph to 1, we force each vertex in `set1` to be matched with at most one vertex in `set2`. The Max Flow will be equal to the maximum number of matchings on the original graph (see Figure 4.42—right for an example).

Max Independent Set and Min Vertex Cover on Bipartite Graph
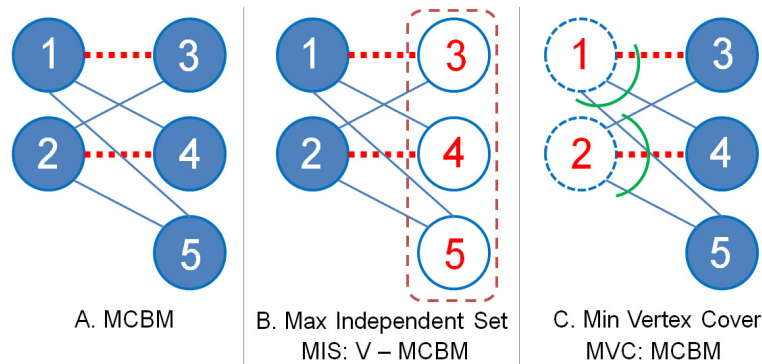
Figure 4.43: MCBM Variants

An Independent Set (IS) of a graph G is a subset of the vertices such that no two vertices in the subset represent an edge of G. A Max IS (MIS) is an IS such that adding any other vertex to the set causes the set to contain an edge. In Bipartite Graph, the size of the MIS + MCBM = V. Or in another word: MIS = V - MCBM. In Figure 4.43.B, we have a Bipartite Graph with 2 vertices on the left side and 3 vertices on the right side. The MCBM is 2 (two dashed lines) and the MIS is 5-2 = 3. Indeed, {3, 4, 5} are the members of the MIS of this Bipartite Graph. Another term for MIS is *Dominating Set*.

A vertex cover of a graph G is a set C of vertices such that each edge of G is incident to at least one vertex in C. In Bipartite Graph, the number of matchings in an MCBM equals the number of vertices in a Min Vertex Cover (MVC)—this is a theorem by a Hungarian mathematician Dénes *König*. In Figure 4.43.C, we have the same Bipartite Graph as earlier with MCBM = 2. The MVC is also 2. Indeed, {1, 2} are the members of the MVC of this Bipartite Graph.

We remark that although the MCBM/MIS/MVC values are unique, the solutions may not be unique. Example: In Figure 4.43.A, we can also match {1, 4} and {2, 5} with the same maximum cardinality of 2.

Sample Application: UVa 12083 - Guardian of Decency

Abridged problem description: Given $N \leq 500$ students (in terms of their height, gender, music style, and favorite sport), determine how many students are eligible for an excursion if the teacher wants any pair of two students satisfy at least one of these four criteria so that no pair of students becomes a couple: 1). Their height differs by more than 40 cm.; 2). They are of the same sex.; 3). Their preferred music style is different.; 4). Their favorite sport is the same (they are likely to be fans of different teams and that would result in fighting).

First, notice that the problem is about finding the Maximum Independent Set, i.e. the chosen students should not have any chance of becoming a couple. Independent Set is a hard problem in general graph, so let's check if the graph is special. Next, notice that there is an easy Bipartite Graph in the problem description: The gender of students (constraint number two). We can put the male students on the left side and the female students on the right side. At this point, we should ask: What should be the edges of this Bipartite Graph? The answer is related to the Independent Set problem: We draw an edge between a male student i and a female student j if there is a chance that (i, j) may become a couple.

In the context of this problem: If i and j have *DIFFERENT* gender *and* their height differs by *NOT MORE* than 40 cm *and* their preferred music style is *THE SAME and* their favorite sport is *DIFFERENT*, then this pair, one male student i and one female student j, has a high probability to be a couple. The teacher can only choose one of them.

Now, once we have this Bipartite Graph, we can run the MCBM algorithm and report: $N - MCBM$. With this example, we again re-highlighted the importance of having good *graph modeling* skill! There is no point knowing MCBM algorithm and its code if contestant cannot identify the Bipartite Graph from the problem description in the first place.

Augmenting Path Algorithm for Max Cardinality Bipartite Matching

There is a better way to solve the MCBM problem in programming contest (in terms of implementation time) rather than going via the 'Max Flow route'. We can use the specialized and easy to implement $O(VE)$ *augmenting path* algorithm. With its implementation handy, all the MCBM problems, including other graph problems that requires MCBM—like the Max Independent Set in Bipartite Graph, Min Vertex Cover in Bipartite Graph, and Min Path Cover on DAG (see Section 9.24)—can be easily solved.

An augmenting path is a path that starts from a *free (unmatched)* vertex on the left set of the Bipartite Graph, alternate between a free edge (now on the right set), a matched edge (now on the left set again), ..., a free edge (now on the right set) until the path finally arrives on a *free vertex* on the right set of the Bipartite Graph. A lemma by Claude *Berge* in 1957 states that a matching M in graph G is maximum (has the max possible number of edges) if and only if there is no more augmenting path in G. This augmenting path algorithm is a direct implementation of Berge's lemma: Find and then eliminate *augmenting paths*.

Now let's take a look at a simple Bipartite Graph in Figure 4.44 with n and m vertices on the left set and the right set, respectively. Vertices on the left set are numbered from [1..n] and vertices of the right set are numbered from [n+1..n+m]. This algorithm tries to find and then eliminates augmenting paths starting from free vertices on the left set.

We start with a free vertex 1. In Figure 4.44.A, we see that this algorithm will 'wrongly[25]' match vertex 1 with vertex 3 (rather than vertex 1 with vertex 4) as path 1-3 is already a simple augmenting path. Both vertex 1 and vertex 3 are free vertices. By matching vertex 1 and vertex 3, we have our first matching. Notice that after we match vertex 1 and 3, we are unable to find another matching.

In the next iteration (when we are in a free vertex 2), this algorithm now shows its full strength by finding the following augmenting path that starts from a free vertex 2 on the left, goes to vertex 3 via a free edge (2-3), goes to vertex 1 via a matched edge (3-1), and finally goes to vertex 4 via a free edge again (1-4). Both vertex 2 and vertex 4 are free vertices. Therefore, the augmenting path is 2-3-1-4 as seen in Figure 4.44.B and 4.44.C.

If we flip the edge status in this augmenting path, i.e. from 'free to matched' and 'matched to free', we will get *one more matching*. See Figure 4.44.C where we flip the status of edges along the augmenting path 2-3-1-4. The updated matching is reflected in Figure 4.44.D.
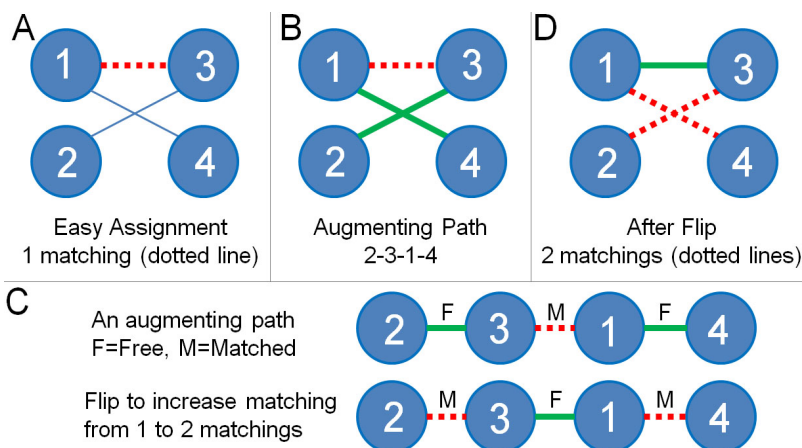
Figure 4.44: Augmenting Path Algorithm

This algorithm will keep doing this process of finding augmenting paths and eliminating them until there is no more augmenting path. As the algorithm repeats $O(E)$ DFS-like[26] code V times, it runs in $O(VE)$. The code is shown below. We remark that this is not the best algorithm for finding MCBM. Later in Section 9.12, we will learn Hopcroft Karp's algorithm that can solve the MCBM problem in $O(\sqrt{V}E)$ [28].

Exercise 4.7.4.1*: In Figure 4.42—right, we have seen a way to reduce an MCBM problem into a Max Flow problem. The question: Does the edges in the flow graph have to be directed? Is it OK if we use undirected edges in the flow graph?

Exercise 4.7.4.2*: List down common keywords that can be used to help contestants spot a bipartite graph in the problem statement! e.g. odd-even, male-female, etc.

Exercise 4.7.4.3*: Suggest a simple improvement for the augmenting path algorithm that can avoid its worst case $O(VE)$ time complexity on (near) complete bipartite graph!

---

[25]We assume that the neighbors of a vertex are ordered based on increasing vertex number, i.e. from vertex 1, we will visit vertex 3 first *before* vertex 4.

[26]To simplify the analysis, we assume that $E > V$ in such bipartite graphs.

```
vi match, vis;                                          // global variables

int Aug(int l) {                      // return 1 if an augmenting path is found
  if (vis[l]) return 0;                                 // return 0 otherwise
  vis[l] = 1;
  for (int j = 0; j < (int)AdjList[l].size(); j++) {
    int r = AdjList[l][j];  // edge weight not needed -> vector<vi> AdjList
    if (match[r] == -1 || Aug(match[r])) {
      match[r] = l; return 1;                           // found 1 matching
  } }
  return 0;                                             // no matching
}
// inside int main()
  // build unweighted bipartite graph with directed edge left->right set
  int MCBM = 0;
  match.assign(V, -1);     // V is the number of vertices in bipartite graph
  for (int l = 0; l < n; l++) {                      // n = size of the left set
    vis.assign(n, 0);                                // reset before each recursion
    MCBM += Aug(l);
  }
  printf("Found %d matchings\n", MCBM);
```

Visualization: `www.comp.nus.edu.sg/~stevenha/visualization/matching.html`

Source code: `ch4_09_mcbm.cpp/java`

Remarks About Special Graphs in Programming Contests

Of the four special graphs mentioned in this Section 4.7. DAGs and Trees are more popular, especially for IOI contestants. It is *not* rare that Dynamic Programming (DP) on DAG or on tree appear as IOI task. As these DP variants (typically) have efficient solutions, the input size for them are usually large. The next most popular special graph is the Bipartite Graph. This special graph is suitable for Network Flow and Bipartite Matching problems. We reckon that contestants must master the usage of the simpler augmenting path algorithm for solving the Max Cardinality Bipartite Matching (MCBM) problem. We have seen in this section that many graph problems are somehow reduce-able to MCBM. ICPC contestants should be familiar with Bipartite Graph on top of DAG and Tree. IOI contestants do not have to worry with Bipartite Graph as it is still outside IOI 2009 syllabus [20]. The other special graph discussed in this chapter—the Eulerian Graph—does not have too many contest problems involving it nowadays. There are other possible special graphs, but we rarely encounter them, e.g. Planar Graph; Complete Graph K_n; Forest of Paths; Star Graph; etc. When they appear, try to utilize their special properties to speed up your algorithms.

Profile of Algorithm Inventors

Dénes König (1884-1944) was a Hungarian mathematician who worked in and wrote the first textbook on the field of graph theory. In 1931, König describes an equivalence between the Maximum Matching problem and the Minimum Vertex Cover problem in the context of Bipartite Graphs, i.e. he proves that MCBM = MVC in Bipartite Graph.

Claude Berge (1926-2002) was a French mathematician, recognized as one of the modern founders of combinatorics and graph theory. His main contribution that is included in this book is the Berge's lemma, which states that a matching M in a graph G is maximum if and only if there is no more augmenting path with respect to M in G.

Programming Exercises related to Special Graphs:

- Single-Source Shortest/Longest Paths on DAG
    1. UVa 00103 - Stacking Boxes (longest paths on DAG; backtracking OK)
    2. **UVa 00452 - Project Scheduling *** (PERT; longest paths on DAG; DP)
    3. UVa 10000 - Longest Paths (longest paths on DAG; backtracking OK)
    4. UVa 10051 - Tower of Cubes (longest paths on DAG; DP)
    5. UVa 10259 - Hippity Hopscotch (longest paths on implicit DAG; DP)
    6. **UVa 10285 - Longest Run ... *** (longest paths on implicit DAG; however, the graph is small enough for recursive backtracking solution)
    7. **UVa 10350 - Liftless Eme *** (shortest paths; implicit DAG; DP)
       Also see: Longest Increasing Subsequence (see Section 3.5.3)

- Counting Paths in DAG
    1. UVa 00825 - Walking on the Safe Side (counting paths in implicit DAG; DP)
    2. UVa 00926 - Walking Around Wisely (similar to UVa 825)
    3. UVa 00986 - How Many? (counting paths in DAG; DP; s: x, y, lastmove, peaksfound; t: try NE/SE)
    4. **UVa 00988 - Many paths, one ... *** (counting paths in DAG; DP)
    5. **UVa 10401 - Injured Queen Problem *** (counting paths in implicit DAG; DP; s: col, row; t: next col, avoid 2 or 3 adjacent rows)
    6. UVa 10926 - How Many Dependencies? (counting paths in DAG; DP)
    7. UVa 11067 - Little Red Riding Hood (similar to UVa 825)
    8. *UVa 11655 - Waterland* (counting paths in DAG and one more similar task: counting the number of vertices involved in the paths)
    9. **UVa 11957 - Checkers *** (counting paths in DAG; DP)

- Converting General Graph to DAG
    1. UVa 00590 - Always on the Run (s: pos, **day_left**)
    2. **UVa 00907 - Winterim Backpack... *** (s: pos, **night_left**)
    3. UVa 00910 - TV Game (s: pos, **move_left**)
    4. UVa 10201 - Adventures in Moving ... (s: pos, **fuel_left**)
    5. UVa 10543 - Traveling Politician (s: pos, **given_speech**)
    6. UVa 10681 - Teobaldo's Trip (s: pos, **day_left**)
    7. UVa 10702 - Traveling Salesman (s: pos, **T_left**)
    8. UVa 10874 - Segments (s: row, **left/right**; t: go left/right)
    9. **UVa 10913 - Walking ... *** (s: r, c, **neg_left**, **stat**; t: down/(left/right))
    10. UVa 11307 - Alternative Arborescence (Min Chromatic Sum, max 6 colors)
    11. **UVa 11487 - Gathering Food *** (s: row, col, **cur_food**, **len**; t: 4 dirs)
    12. UVa 11545 - Avoiding ... (s: cPos, **cTime**, **cWTime**; t: move forward/rest)
    13. UVa 11782 - Optimal Cut (s: id, **rem_K**; t: take root/try left-right subtree)
    14. SPOJ 0101 - Fishmonger (discussed in this section)

- Tree

    1. UVa 00112 - Tree Summing (backtracking)
    2. UVa 00115 - Climbing Trees (tree traversal, Lowest Common Ancestor)
    3. UVa 00122 - Trees on the level (tree traversal)
    4. UVa 00536 - Tree Recovery (reconstructing tree from pre + inorder)
    5. *UVa 00548 - Tree* (reconstructing tree from in + postorder traversal)
    6. UVa 00615 - Is It A Tree? (graph property check)
    7. UVa 00699 - The Falling Leaves (preorder traversal)
    8. UVa 00712 - S-Trees (simple binary tree traversal variant)
    9. UVa 00839 - Not so Mobile (can be viewed as recursive problem on tree)
    10. UVa 10308 - Roads in the North (diameter of tree, discussed in this section)
    11. **UVa 10459 - The Tree Root *** (identify the diameter of this tree)
    12. UVa 10701 - Pre, in and post (reconstructing tree from pre + inorder)
    13. **UVa 10805 - Cockroach Escape ... *** (involving diameter)
    14. *UVa 11131 - Close Relatives* (read tree; produce two postorder traversals)
    15. *UVa 11234 - Expressions* (converting post-order to level-order, binary tree)
    16. UVa 11615 - Family Tree (counting size of subtrees)
    17. **UVa 11695 - Flight Planning *** (cut the worst edge along the tree diameter, link two centers)
    18. *UVa 12186 - Another Crisis* (the input graph is a tree)
    19. *UVa 12347 - Binary Search Tree* (given pre-order traversal of a BST, use BST property to get the BST, output the post-order traversal that BST)

- Eulerian Graph

    1. UVa 00117 - The Postal Worker ... (Euler tour, cost of tour)
    2. UVa 00291 - The House of Santa ... (Euler tour, small graph, backtracking)
    3. **UVa 10054 - The Necklace *** (printing the Euler tour)
    4. UVa 10129 - Play on Words (Euler Graph property check)
    5. **UVa 10203 - Snow Clearing *** (the underlying graph is Euler graph)
    6. **UVa 10596 - Morning Walk *** (Euler Graph property check)

- Bipartite Graph:

    1. *UVa 00663 - Sorting Slides* (try disallowing an edge to see if MCBM changes; which implies that the edge has to be used)
    2. UVa 00670 - The Dog Task (MCBM)
    3. UVa 00753 - A Plug for Unix (initially a non standard matching problem but this problem can be reduced to a simple MCBM problem)
    4. UVa 01194 - Machine Schedule (LA 2523, Beijing02, Min Vertex Cover/MVC)
    5. UVa 10080 - Gopher II (MCBM)
    6. **UVa 10349 - Antenna Placement *** (Max Independent Set: V - MCBM)
    7. **UVa 11138 - Nuts and Bolts *** (pure MCBM problem, if you are new with MCBM, it is good to start from this problem)
    8. **UVa 11159 - Factors and Multiples *** (MIS, but ans is the MCBM)
    9. *UVa 11419 - SAM I AM* (MVC, König theorem)
    10. UVa 12083 - Guardian of Decency (LA 3415, NorthwesternEurope05, MIS)
    11. UVa 12168 - Cat vs. Dog (LA 4288, NorthwesternEurope08, MIS)
    12. Top Coder Open 2009: Prime Pairs (discussed in this section)

4.8 Solution to Non-Starred Exercises

Exercise 4.2.2.1: Simply replace `dfs(0)` with `bfs` from source `s = 0`.

Exercise 4.2.2.2: Adjacency Matrix, Adjacency List, and Edge List require $O(V)$, $O(k)$, and $O(E)$ to enumerate the list of neighbors of a vertex, respectively (note: k is the number of actual neighbors of a vertex). Since DFS and BFS explores all outgoing edges of each vertex, it's runtime depends on the underlying graph data structure speed in enumerating neighbors. Therefore, the time complexity of DFS and BFS are $O(V \times V = V^2)$, $O(max(V, V \sum_{i=0}^{V-1} k_i) = V + E)$, and $O(V \times E = VE)$ to traverse graph stored in an Adjacency Matrix, Adjacency List, and Edge List, respectively. As Adjacency List is the most efficient data structure for graph traversal, it may be beneficial to convert Adjacency Matrix or Edge List to Adjacency List first (see **Exercise 2.4.1.2***) before traversing the graph.

Exercise 4.2.3.1: Start with disjoint vertices. For each `edge(u, v)`, do `unionSet(u, v)`. The state of disjoint sets after processing all edges represent the connected components. BFS solution is 'trivial': Simply change `dfs(i)` to `bfs(i)`. Both run in $O(V + E)$.

Exercise 4.2.5.1: This is a kind of 'post-order traversal' in binary tree traversal terminology. Function `dfs2` visits all the children of u before appending vertex u at the back of vector `ts`. This satisfies the topological sort property!

Exercise 4.2.5.2: The answer is to use a Linked List. However, since in Chapter 2, we have said that we want to avoid using Linked List, we decide to use `vi ts` here.

Exercise 4.2.5.3: The algorithm will still terminate, but the output is now irrelevant as a non DAG has no topological sort.

Exercise 4.2.5.4: We must use recursive backtracking to do so.

Exercise 4.2.6.3: Proof by contradiction. Assume that a Bipartite Graph has an odd (length) cycle. Let the odd cycle contains $2k + 1$ vertices for a certain integer k that forms this path: $v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow ... \rightarrow v_{2k-1} \rightarrow v_{2k} \rightarrow v_0$. Now, we can put v_0 in the left set, v_1 in the right set, ..., v_{2k} on the left set again, but then we have edge (v_{2k}, v_0) that is not in the left set. This is not a cycle $\rightarrow$ contradiction. Therefore, a Bipartite Graph has no odd cycle. This property can be important to solve some problems involving Bipartite Graph.

Exercise 4.2.7.1: Two back edges: $2 \rightarrow 1$ and $6 \rightarrow 4$.

Exercise 4.2.8.1: Articulation points: 1, 3 and 6; Bridges: 0-1, 3-4, 6-7, and 6-8.

Exercise 4.2.9.1: Proof by contradiction. Assume that there exists a path from vertex u to w and w to v where w is outside the SCC. From this, we can conclude that we can travel from vertex w to any vertices in the SCC and from any vertices in the SCC to w. Therefore, vertex w should be in the SCC. Contradiction. So there is no path between two vertices in an SCC that ever leaves the SCC.

Exercise 4.3.2.1: We can stop when the number of disjoint sets is already one. The simple modification: Change the start of the MST loop from: `for (int i = 0; i < E; i++) {` To: `for (int i = 0; i < E && disjointSetSize > 1; i++) {` Alternatively, we count the number of edges taken so far. Once it hits $V - 1$, we can stop.

Exercise 4.3.4.1: We found that MS 'Forest' and Second Best ST problems are harder to be solved with Prim's algorithm.

Exercise 4.4.2.1: For this variant, the solution is easy. Simply enqueue all the sources and set `dist[s] = 0` for all the sources before running the BFS loop. As this is just one BFS call, it runs in $O(V + E)$.

Exercise 4.4.2.2: At the start of the while loop, when we pop up the front most vertex from the queue, we check if that vertex is the destination. If it is, we break the loop there. The worst time complexity is still $O(V + E)$ but our BFS will stop sooner if the destination vertex is close to the source vertex.

Exercise 4.4.2.3: You can transform that constant-weighted graph into an unweighted graph by replacing all edge weights with ones. The SSSP information obtained by BFS is then multiplied with the constant C to get the actual answers.

Exercise 4.4.3.1: On positive weighted graph, yes. Each vertex will only be processed once. Each time a vertex is processed, we try to relax its neighbors. Because of lazy deletion, we may have at most $O(E)$ items in the priority queue at a certain time, but this is still $O(\log E) = O(\log V^2) = O(2 \times \log V) = O(\log V)$ per each dequeue or enqueue operations. Thus, the time complexity remains at $O((V + E) \log V)$. On graph with (a few) negative weight edges but no negative cycle, it runs slower due to the need of re-processing processed vertices but the shortest paths values are correct (unlike the Dijkstra's implementation shown in [7]). This is shown in an example in Section 4.4.4. On rare cases, this Dijkstra's implementation can run very slow on certain graph with some negative weight edges although the graph has no negative cycle (see **Exercise 4.4.3.2***). If the graph has negative cycle, this Dijkstra's implementation variant will be trapped in an infinite loop.

Exercise 4.4.3.3: Use `set<ii>`. This set stores sorted pair of vertex information as shown in Section 4.4.3. Vertex with the minimum distance is the first element in the (sorted) set. To update the distance of a certain vertex from source, we search and then delete the old value pair. Then we insert a new value pair. As we process each vertex and edge once and each time we access `set<ii>` in $O(\log V)$, the overall time complexity of Dijkstra's implementation variant using `set<ii>` is still $O((V + E) \log V)$.

Exercise 4.4.3.4: In Section 2.3, we have shown the way to reverse the default max heap of C++ STL `priority_queue` into a min heap by multiplying the sort keys with -1.

Exercise 4.4.3.5: Similar answer as with **Exercise 4.4.2.2** if the given weighted graph has no negative weight edge. There is a potential for wrong answer if the given weighted graph has negative weight edge.

Exercise 4.4.3.6: No, we cannot use DP. The state and transition modeling outlined in Section 4.4.3 creates a State-Space graph that is *not* a DAG. For example, we can start from state $(s, 0)$, add 1 unit of fuel at vertex s to reach state $(s, 1)$, go to a neighbor vertex y—suppose it is just 1 unit distance away—to reach state $(y, 0)$, add 1 unit of fuel again at vertex y to reach state $(y, 1)$, and then return back to state $(s, 0)$ (a cycle). So, this problem is a shortest path problem on general weighted graph. We need to use Dijkstra's algorithm.

Exercise 4.4.4.1: This is because initially only the source vertex has the correct distance information. Then, every time we relax all E edges, we guarantee that at least one more vertex with one more hop (in terms of edges used in the shortest path from source) has the correct distance information. In **Exercise 4.4.1.1**, we have seen that the shortest path must be a simple path (has at most $E = V - 1$ edges. So, after $V - 1$ pass of Bellman Ford's, even the vertex with the largest number of hops will have the correct distance information.

Exercise 4.4.4.2: Put a boolean flag `modified = false` in the outermost loop (the one that repeats all E edges relaxation $V - 1$ times). If at least one relaxation operation is done in the inner loops (the one that explores all E edges), set `modified = true`. Immediately break the outermost loop if `modified` is still false after all E edges have been examined. If this no-relaxation happens at the outermost loop iteration i, there will be no further relaxation in iteration $i + 1, i + 2, \ldots, i = V - 1$ either.

Exercise 4.5.1.1: This is because we will add `AdjMat[i][k]` + `AdjMat[k][j]` which will *overflow* if both `AdjMat[i][k]` and `AdjMat[k][j]` are near the MAX_INT range, thus giving wrong answer.

Exercise 4.5.1.2: Floyd Warshall's works in graph with negative weight edges. For graph with negative cycle, see Section 4.5.3 about 'finding negative cycle'.

Exercise 4.5.3.1: Running Warshall's algorithm directly on a graph with $V \leq 1000$ will result in TLE. Since the number of queries is low, we can afford to run $O(V + E)$ DFS per query to check if vertex u and v are connected by a path. If the input graph is directed, we can find the SCCs of the directed graphs first in $O(V + E)$. If u and v belong to the same SCC, then u will surely reach v. This can be tested with no additional cost. If SCC that contains u has a directed edge to SCC that contains v, then u will also reach v. But the connectivity check between different SCCs is much harder to check and we may as well just use a normal DFS to get the answer.

Exercise 4.5.3.3: In Floyd Warshall's, replace addition with multiplication and set the main diagonal to 1.0. After we run Floyd Warshall's, we check if the main diagonal > 1.0.

Exercise 4.6.3.1: A. 150; B = 125; C = 60.

Exercise 4.6.3.2: In the updated code below, we use *both* Adjacency List (for fast enumeration of neighbors; do not forget to include backward edges due to backward flow) and Adjacency Matrix (for fast access to residual capacity) of the same flow graph, i.e. we concentrate on improving this line: `for (int v = 0; v < MAX_V; v++)`. We also replace `vi dist(MAX_V, INF);` to `bitset<MAX_V> visited` to speed up the code a little bit more.

```
// inside int main(), assume that we have both res (AdjMatrix) and AdjList
  mf = 0;
  while (1) {                      // now a true O(VE^2) Edmonds Karp's algorithm
    f = 0;
    bitset<MAX_V> vis; vis[s] = true;      // we change vi dist to bitset!
    queue<int> q; q.push(s);
    p.assign(MAX_V, -1);
    while (!q.empty()) {
      int u = q.front(); q.pop();
      if (u == t) break;
      for (int j = 0; j < (int)AdjList[u].size(); j++) {   // AdjList here!
        int v = AdjList[u][j];                    // we use vector<vi> AdjList
        if (res[u][v] > 0 && !vis[v])
          vis[v] = true, q.push(v), p[v] = u;
      }
    }
    augment(t, INF);
    if (f == 0) break;
    mf += f;
  }
```

Exercise 4.6.4.1: We use ∞ for the capacity of the 'middle directed edges' between the left and the right sets of the bipartite graph for the overall correctness of this flow graph modeling. If the capacities from the right set to sink t is *not* 1 as in UVa 259, we will get wrong Max Flow value if we set the capacity of these 'middle directed edges' to 1.

4.9   Chapter Notes

We end this relatively long chapter by making a remark that this chapter has lots of algorithms and algorithm inventors—the most in this book. This trend will likely increase in the future, i.e. there will be *more* graph algorithms. However, we have to warn the contestants that recent ICPCs and IOIs usually do not just ask contestants to solve problems involving the pure form of these graph algorithms. New problems usually require contestants to use creative graph modeling, combine two or more algorithms or to combine an algorithm with some advanced data structures, e.g. combining the longest path in DAG with Segment Tree data structure; using SCC contraction of Directed Graph to transform the graph into DAG before solving the actual problem on DAG; etc. These harder forms of graph problems are discussed in Section 8.4.

This chapter, albeit already quite long, still omits many known graph algorithms and graph problems that may be tested in ICPCs, namely: k-th shortest paths, Bitonic Traveling Salesman Problem (see Section 9.2), **Chu Liu Edmonds algorithm** for Min Cost Arborescence problem, **Hopcroft Karp's** MCBM algorithm (see Section 9.12), **Kuhn Munkres's (Hungarian)** weighted MCBM algorithm, **Edmonds's Matching** algorithm for general graph, etc. We invite readers to check Chapter 9 for some of these algorithms.

If you want to increase your winning chance in ACM ICPC, please spend some time to study more graph algorithms/problems beyond[27] this book. These harder ones rarely appears in *regional* contests and if they are, they usually become the *decider* problem. Harder graph problems are more likely to appear in the ACM ICPC World Finals level.

However, we have good news for IOI contestants. We believe that most graph materials in the IOI syllabus are already covered in this chapter. You need to master the basic algorithms covered in this chapter and then improve your problem solving skills in applying these basic algorithms to creative graph problems frequently posed in IOI.

Statistics	First Edition	Second Edition	Third Edition
Number of Pages	35	49 (+40%)	70 (+43%)
Written Exercises	8	30 (+275%)	30+20*=50 (+63%)
Programming Exercises	173	230 (+33%)	248 (+8%)

The breakdown of the number of programming exercises from each section is shown below:

Section	Title	Appearance	% in Chapter	% in Book
4.2	**Graph Traversal**	65	26%	4%
4.10	Minimum Spanning Tree	25	10%	1%
4.4	Single-Source Shortest Paths	51	21%	3%
4.5	All-Pairs Shortest Paths	27	11%	2%
4.6	Network Flow	13	5%	1%
4.7	**Special Graphs**	67	27%	4%

---

[27]Interested readers are welcome to explore Felix's paper [23] that discusses maximum flow algorithm for *large* graphs of 411 million vertices and 31 billion edges!

Chapter 5

Mathematics

> *We all use math every day; to predict weather, to tell time, to handle money.*
> *Math is more than formulas or equations; it's logic, it's rationality,*
> *it's using your mind to solve the biggest mysteries we know.*
> — **TV show NUMB3RS**

5.1 Overview and Motivation

The appearance of mathematics-related problems in programming contests is not surprising since Computer Science is deeply rooted in Mathematics. The term 'computer' itself comes from the word 'compute' as computer is built primarily to help human compute numbers. Many interesting real life problems can be modeled as mathematics problems as you will frequently see in this chapter.

Recent ICPC problem sets (especially in Asia) usually contain one or two mathematics problems. Recent IOIs usually do not contain *pure* mathematics tasks, but many tasks do require mathematical insights. This chapter aims to prepare contestants in dealing with many of these mathematics problems.

We are aware that different countries have different emphasis in mathematics training in pre-University education. Thus, some contestants are familiar with the mathematical terms listed in Table 5.1. But for others, these mathematical terms do not ring any bell. Perhaps because the contestant has not learnt it before, or perhaps the term is different in the contestant's native language. In this chapter, we want to make a more level-playing field for the readers by listing as many common mathematical terminologies, definitions, problems, and algorithms that frequently appear in programming contests.

Arithmetic Progression	Geometric Progression	Polynomial
Algebra	Logarithm/Power	BigInteger
Combinatorics	Fibonacci	Golden Ratio
Binet's formula	Zeckendorf's theorem	Catalan Numbers
Factorial	Derangement	Binomial Coefficients
Number Theory	Prime Number	Sieve of Eratosthenes
Modified Sieve	Miller-Rabin's	Euler Phi
Greatest Common Divisor	Lowest Common Multiple	Extended Euclid
Linear Diophantine Equation	Cycle-Finding	Probability Theory
Game Theory	Zero-Sum Game	Decision Tree
Perfect Play	Minimax	Nim Game

Table 5.1: List of *some* mathematical terms discussed in this chapter

5.2 Ad Hoc Mathematics Problems

We start this chapter with something light: The Ad Hoc mathematics problems. These are programming contest problems that require no more than basic programming skills and some fundamental mathematics. As there are still too many problems in this category, we further divide them into sub-categories, as shown below. These problems are not placed in Section 1.4 as they are Ad Hoc problems with mathematical flavor. You can actually jump from Section 1.4 to this section if you prefer to do so. But remember that many of these problems are the easier ones. To do well in the actual programming contests, contestants must also master *the other sections* of this chapter.

- The Simpler Ones—just a few lines of code per problem to boost confidence. These problems are for those who have not solved any mathematics-related problems before.

- Mathematical Simulation (Brute Force)
  The solutions to these problems can be obtained by simulating the mathematical process. Usually, the solution requires some form of loops. Example: Given a set S of $1M$ random integers and an integer X. How many integers in S are less than X? Answer: Brute force, scan all the $1M$ integers and count how many of them are less than X. This is slightly faster than sorting the $1M$ integers first. See Section 3.2 if you need to review various (iterative) Complete Search/brute force techniques. Some mathematical problems solvable with brute force approach are also listed in that Section 3.2.

- Finding Pattern or Formula
  These problems require the problem solver to read the problem description carefully to spot the pattern or simplified formula. Attacking them directly will usually result in TLE verdict. The actual solutions are usually short and do not require loops or recursions. Example: Let set S be an infinite set of *square integers* sorted in increasing order: $\{1, 4, 9, 16, 25, \dots\}$. Given an integer X ($1 \leq X \leq 10^{17}$), determine how many integers in S are less than X? Answer: $\lfloor\sqrt{X-1}\rfloor$.

- Grid
  These problems involve grid manipulation. The grid can be complex, but the grid follow some primitive rules. The 'trivial' 1D/2D grid are not classified here. The solution usually depends on the problem solver's creativity on finding the patterns to manipulate/navigate the grid or in converting the given one into a simpler one.

- Number Systems or Sequences
  Some Ad Hoc mathematics problems involve definitions of existing (or fictional) Number Systems or Sequences and our task is to produce either the number (sequence) within some range or the n-th one, verify if the given number (sequence) is valid according to definition, etc. Usually, following the problem description carefully is the key to solving the problem. But some harder problems require us to simplify the formula first. Some well-known examples are:

  1. Fibonacci numbers (Section 5.4.1): 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...

  2. Factorial (Section 5.5.3): 1, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880, ...

  3. Derangement (Section 9.8): 1, 0, 1, 2, 9, 44, 265, 1854, 14833, 133496, ...

  4. Catalan numbers (Section 5.4.3): 1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, ...

5. Arithmetic progression series: a_1, $(a_1 + d)$, $(a_1 + 2 \times d)$, $(a_1 + 3 \times d)$, ..., e.g. 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, ... that starts with $a_1 = 1$ and with difference of $d = 1$ between consecutive terms. The sum of the first n terms of this arithmetic progression series $S_n = \frac{n}{2} \times (2 \times a_1 + (n-1) \times d)$.

6. Geometric progression series, e.g. a_1, $a_1 \times r$, $a_1 \times r^2$, $a_1 \times r^3$, ..., e.g. 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, ... that starts with $a_1 = 1$ and with common ratio $r = 2$ between consecutive terms. The sum of the first n terms of this geometric progression series $S_n = a \times \frac{1-r^n}{1-r}$.

- Logarithm, Exponentiation, Power
  These problems involve the (clever) usage of `log()` and/or `exp()` function.
  Some of the important ones are shown in the written exercises below.

- Polynomial
  These problems involve polynomial evaluation, derivation, multiplication, division, etc. We can represent a polynomial by storing the coefficients of the polynomial's terms sorted by their powers (usually in descending order). The operations on polynomial usually require some careful usage of loops.

- Base Number Variants
  These are the mathematical problems involving base number, but they are not the *standard* conversion problem that can be easily solved with Java BigInteger technique (see Section 5.3).

- Just Ad Hoc
  These are other mathematics-related problems that cannot be classified yet as one of the sub-categories above.

We suggest that the readers—especially those who are new with mathematics problems— kick start their training programme on mathematics problems by solving at least 2 or 3 problems *from each sub-category*, especially the ones that we highlight as **must try ***.

Exercise 5.2.1: What should we use in C/C++/Java to compute $\log_b$(a) (base b)?

Exercise 5.2.2: What will be returned by `(int)floor(1 + log10((double)a))`?

Exercise 5.2.3: How to compute $\sqrt[n]{a}$ (the n-th root of a) in C/C++/Java?

Exercise 5.2.4*: Study the (Ruffini-)Horner's method for finding the roots of a polynomial equation $f(x) = 0$!

Exercise 5.2.5*: Given $1 < a < 10, 1 \le n \le 100000$, show how to compute the value of $1 \times a + 2 \times a^2 + 3 \times a^3 + \ldots + n \times a^n$ efficiently, i.e. in $O(\log n)$!

Programming Exercises related to Ad Hoc Mathematics problems:

- The Simpler Ones

  1. UVa 10055 - Hashmat the Brave Warrior (absolute function; use long long)
  2. UVa 10071 - Back to High School ... (super simple: outputs $2 \times v \times t$)
  3. UVa 10281 - Average Speed (distance = speed $\times$ time elapsed)

4. UVa 10469 - To Carry or not to Carry (super simple if you use `xor`)
5. **UVa 10773 - Back to Intermediate ... *** (several tricky cases)
6. UVa 11614 - Etruscan Warriors Never ... (find roots of a quadratic equation)
7. **UVa 11723 - Numbering Road *** (simple math)
8. UVa 11805 - Bafana Bafana (very simple $O(1)$ formula exists)
9. **UVa 11875 - Brick Game *** (get median of a sorted input)
10. *UVa 12149 - Feynman* (finding the pattern; square numbers)
11. *UVa 12502 - Three Families* (must understand the 'wording trick' first)

- Mathematical Simulation (Brute Force), Easier

1. UVa 00100 - The 3n + 1 problem (do as asked; note that j can be $< i$)
2. UVa 00371 - Ackermann Functions (similar to UVa 100)
3. **UVa 00382 - Perfection *** (do trial division)
4. UVa 00834 - Continued Fractions (do as asked)
5. UVa 00906 - Rational Neighbor (compute c, from $d = 1$ until $\frac{a}{b} < \frac{c}{d}$)
6. **UVa 01225 - Digit Counting *** (LA 3996, Danang07, N is small)
7. UVa 10035 - Primary Arithmetic (count the number of carry operations)
8. **UVa 10346 - Peter's Smoke *** (interesting simulation problem)
9. UVa 10370 - Above Average (compute average, see how many are above it)
10. UVa 10783 - Odd Sum (input range is very small, just brute force it)
11. UVa 10879 - Code Refactoring (just use brute force)
12. UVa 11150 - Cola (similar to UVa 10346, be careful with boundary cases!)
13. UVa 11247 - Income Tax Hazard (brute force around the answer to be safe)
14. UVa 11313 - Gourmet Games (similar to UVa 10346)
15. UVa 11689 - Soda Surpler (similar to UVa 10346)
16. UVa 11877 - The Coco-Cola Store (similar to UVa 10346)
17. UVa 11934 - Magic Formula (just do plain brute-force)
18. *UVa 12290 - Counting Game* (no '-1' in the answer)
19. *UVa 12527 - Different Digits* (try all, check repeated digits)

- Mathematical Simulation (Brute Force), Harder

1. *UVa 00493 - Rational Spiral* (simulate the spiral process)
2. *UVa 00550 - Multiplying by Rotation* (rotamult property; try one by one starting from 1 digit)
3. **UVa 00616 - Coconuts, Revisited *** (brute force up to $\sqrt{n}$, get pattern)
4. *UVa 00697 - Jack and Jill* (requires some output formatting and basic knowledge about Physics)
5. UVa 00846 - Steps (uses the sum of arithmetic progression formula)
6. *UVa 10025 - The ? 1 ? 2 ? ...* (simplify the formula first, iterative)
7. *UVa 10257 - Dick and Jane* (we can brute force the integer ages of spot, puff, and yertle; need some mathematical insights)
8. *UVa 10624 - Super Number* (backtracking with divisibility check)
9. **UVa 11130 - Billiard bounces *** (use billiard table reflection technique: mirror the billiard table to the right (and/or top) so that we will only deal with one straight line instead of bouncing lines)
10. ***UVa 11254 - Consecutive Integers **** (use sum of arithmetic progression: $n = \frac{r}{2} \times (2 \times a + r - 1)$ or $a = (2 \times n + r - r^2)/(2 \times r)$; as n is given, brute force all values of r from $\sqrt{2n}$ down to 1, stop at the first valid a)
11. UVa 11968 - In The Airport (average; fabs; if ties, choose the smaller one!) Also see some mathematical problems in Section 3.2.

- Finding Pattern or Formula, Easier

  1. UVa 10014 - Simple calculations (derive the required formula)
  2. UVa 10170 - The Hotel with Infinite ... (one liner formula exists)
  3. UVa 10499 - The Land of Justice (simple formula exists)
  4. UVa 10696 - f91 (very simple formula simplification)
  5. **_UVa 10751 - Chessboard *_** (trival for $N = 1$ and $N = 2$; derive the formula first for $N > 2$; hint: use diagonal as much as possible)
  6. **UVa 10940 - Throwing Cards Away II *** (find pattern with brute force)
  7. UVa 11202 - The least possible effort (consider symmetry and flip)
  8. **_UVa 12004 - Bubble Sort *_** (try small n; get the pattern; use long long)
  9. *UVa 12027 - Very Big Perfect Square* (sqrt trick)

- Finding Pattern or Formula, Harder

  1. *UVa 00651 - Deck* (use the given sample I/O to derive the simple formula)
  2. UVa 00913 - Joana and The Odd ... (derive the short formulas)
  3. **UVa 10161 - Ant on a Chessboard *** (involves sqrt, ceil...)
  4. *UVa 10493 - Cats, with or without Hats* (tree, derive the formula)
  5. UVa 10509 - R U Kidding Mr. ... (there are only three different cases)
  6. UVa 10666 - The Eurocup is here (analyze the binary representation of X)
  7. UVa 10693 - Traffic Volume (derive the short Physics formula)
  8. *UVa 10710 - Chinese Shuffle* (the formula is a bit hard to derive; involving modPow; see Section 5.3 or Section 9.21)
  9. *UVa 10882 - Koerner's Pub* (inclusion-exclusion principle)
  10. UVa 10970 - Big Chocolate (direct formula exists, or use DP)
  11. UVa 10994 - Simple Addition (formula simplification)
  12. **UVa 11231 - Black and White Painting *** (there is $O(1)$ formula)
  13. *UVa 11246 - K-Multiple Free Set* (derive the formula)
  14. UVa 11296 - Counting Solutions to an ... (simple formula exists)
  15. *UVa 11298 - Dissecting a Hexagon* (simple maths; derive the pattern first)
  16. *UVa 11387 - The 3-Regular Graph* (impossible for odd n or when $n = 2$; if n is a multiple of 4, consider complete graph K_4; if $n = 6 + k \times 4$, consider one 3-Regular component of 6 vertices and the rest are K_4 as in previous case)
  17. *UVa 11393 - Tri-Isomorphism* (draw several small K_n, derive the pattern)
  18. **_UVa 11718 - Fantasy of a Summation *_** (convert loops to a closed form formula, use `modPow` to compute the results, see Section 5.3 and 9.21)

- Grid

  1. **UVa 00264 - Count on Cantor *** (math, grid, pattern)
  2. UVa 00808 - Bee Breeding (math, grid, similar to UVa 10182)
  3. UVa 00880 - Cantor Fractions (math, grid, similar to UVa 264)
  4. **UVa 10182 - Bee Maja *** (math, grid)
  5. **UVa 10233 - Dermuba Triangle *** (the number of items in row forms arithmetic progression series; use hypot)
  6. UVa 10620 - A Flea on a Chessboard (just simulate the jumps)
  7. UVa 10642 - Can You Solve It? (the reverse of UVa 264)
  8. *UVa 10964 - Strange Planet* (convert the coordinates to (x, y), then this problem is just about finding Euclidean distance between two coordinates)

9. *SPOJ 3944 - Bee Walk* (a grid problem)

- Number Systems or Sequences

    1. UVa 00136 - Ugly Numbers (use similar technique as UVa 443)
    2. UVa 00138 - Street Numbers (arithmetic progression formula, precalculated)
    3. UVa 00413 - Up and Down Sequences (simulate; array manipulation)
    4. **<u>UVa 00443 - Humble Numbers *</u>** (try all $2^i \times 3^j \times 5^k \times 7^l$, sort)
    5. UVa 00640 - Self Numbers (DP bottom up, generate the numbers, flag once)
    6. UVa 00694 - The Collatz Sequence (similar to UVa 100)
    7. UVa 00962 - Taxicab Numbers (pre-calculate the answer)
    8. UVa 00974 - Kaprekar Numbers (there are not that many Kaprekar numbers)
    9. UVa 10006 - Carmichael Numbers (non prime which has ≥ 3 prime factors)
    10. **<u>UVa 10042 - Smith Numbers *</u>** (prime factorization, sum the digits)
    11. *UVa 10049 - Self-describing Sequence* (enough to get past $> 2G$ by storing only the first $700K$ numbers of the Self-describing sequence)
    12. UVa 10101 - Bangla Numbers (follow the problem description carefully)
    13. **<u>UVa 10408 - Farey Sequences *</u>** (first, generate `(i, j)` pairs such that `gcd(i, j) = 1`, then sort)
    14. UVa 10930 - A-Sequence (ad-hoc, follow the rules given in description)
    15. *UVa 11028 - Sum of Product* (this is 'dartboard sequence')
    16. UVa 11063 - B2 Sequences (see if a number is repeated, be careful with -ve)
    17. UVa 11461 - Square Numbers (answer is $\sqrt{b} - \sqrt{a-1}$)
    18. UVa 11660 - Look-and-Say sequences (simulate, break after j-th character)
    19. UVa 11970 - Lucky Numbers (square numbers, divisibility check, bf)

- Logarithm, Exponentiation, Power

    1. UVa 00107 - The Cat in the Hat (use logarithm, power)
    2. UVa 00113 - Power Of Cryptography (use `exp(ln(x)` $\times$ `y)`)
    3. UVa 00474 - Heads Tails Probability (this is just a `log` & `pow` exercise)
    4. UVa 00545 - Heads (use logarithm, power, similar to UVa 474)
    5. **<u>UVa 00701 - Archaelogist's Dilemma *</u>** (use log to count # of digits)
    6. *UVa 01185 - BigNumber* (number of digits of factorial, use logarithm to solve it; $\log(n!) = \log(n \times (n-1) \ldots \times 1) = \log(n) + \log(n-1) + \ldots + \log(1)$)
    7. **<u>UVa 10916 - Factstone Benchmark *</u>** (use logarithm, power)
    8. *UVa 11384 - Help is needed for Dexter* (finding the smallest power of two greater than n, can be solved easily using $ceil(eps + \log_2(n))$)
    9. *UVa 11556 - Best Compression Ever* (related to power of two, use long long)
    10. UVa 11636 - Hello World (uses logarithm)
    11. UVa 11666 - Logarithms (find the formula!)
    12. UVa 11714 - Blind Sorting (use decision tree model to find min and second min; eventually the solution only involves logarithm)
    13. **<u>UVa 11847 - Cut the Silver Bar *</u>** ($O(1)$ math formula exists: $\lfloor \log_2(n) \rfloor$)
    14. UVa 11986 - Save from Radiation ($\log_2(N+1)$; manual check for precision)
    15. *UVa 12416 - Excessive Space Remover* (the answer is $\log_2$ of the max consecutive spaces in a line)

- Polynomial

  1. *UVa 00126 - The Errant Physicist* (polynomial multiplication and tedious output formatting)
  2. UVa 00392 - Polynomial Showdown (follow the orders: output formatting)
  3. **UVa 00498 - Polly the Polynomial *** (polynomial evaluation)
  4. *UVa 10215 - The Largest/Smallest Box* (two trivial cases for smallest; derive the formula for largest which involves quadratic equation)
  5. **UVa 10268 - 498' *** (polynomial derivation; Horner's rule)
  6. UVa 10302 - Summation of Polynomials (use long double)
  7. *UVa 10326 - The Polynomial Equation* (given roots of the polynomial, reconstruct the polynomial; formatting)
  8. **UVa 10586 - Polynomial Remains *** (division; manipulate coefficients)
  9. UVa 10719 - Quotient Polynomial (polynomial division and remainder)
  10. *UVa 11692 - Rain Fall* (use algebraic manipulation to derive a quadratic equation; solve it; special case when $H < L$)

- Base Number Variants

  1. **UVa 00377 - Cowculations *** (base 4 operations)
  2. **UVa 00575 - Skew Binary *** (base modification)
  3. UVa 00636 - Squares (base number conversion up to base 99; Java BigInteger cannot be used as it is MAX_RADIX is limited to 36)
  4. UVa 10093 - An Easy Problem (try all)
  5. UVa 10677 - Base Equality (try all from r2 to r1)
  6. **UVa 10931 - Parity *** (convert decimal to binary, count number of '1's)
  7. UVa 11005 - Cheapest Base (try all possible bases from 2 to 36)
  8. UVa 11121 - Base -2 (search for the term 'negabinary')
  9. *UVa 11398 - The Base-1 Number System* (just follow the new rules)
  10. *UVa 12602 - Nice Licence Plates* (simple base conversion)
  11. *SPOJ 0739 - The Moronic Cowmpouter* (find the representation in base -2)
  12. IOI 2011 - Alphabets (practice task; use the more space-efficient base 26)

- Just Ad Hoc

  1. UVa 00276 - Egyptian Multiplication (multiplication of Egyptian hieroglyphs)
  2. UVa 00496 - Simply Subsets (set manipulation)
  3. *UVa 00613 - Numbers That Count* (analyze the number; determine the type; similar spirit with the cycle finding problem in Section 5.7)
  4. **UVa 10137 - The Trip *** (be careful with precision error)
  5. UVa 10190 - Divide, But Not Quite ... (simulate the process)
  6. *UVa 11055 - Homogeneous Square* (not classic, observation needed to avoid brute-force solution)
  7. *UVa 11241 - Humidex* (the hardest case is computing Dew point given temperature and Humidex; derive it with Algebra)
  8. **UVa 11526 - H(n) *** (brute force up to $\sqrt{n}$, find the pattern, avoid TLE)
  9. UVa 11715 - Car (physics simulation)
  10. UVa 11816 - HST (simple math, precision required)
  11. **_UVa 12036 - Stable Grid *_** (use pigeon hole principle)

5.3 Java BigInteger Class

5.3.1 Basic Features

When the intermediate and/or the final result of an integer-based mathematics computation cannot be stored inside the largest built-in integer data type and the given problem cannot be solved with any prime-power factorization (Section 5.5.5) or modulo arithmetic techniques (Section 5.5.8), we have no choice but to resort to BigInteger (a.k.a bignum) libraries. An example: Compute the *precise value* of 25! (the factorial of 25). The result is 15,511,210,043,330,985,984,000,000 (26 digits). This is clearly too large to fit in 64-bit C/C++ `unsigned long long` (or Java `long`).

One way to implement BigInteger library is to store the BigInteger as a (long) string[1]. For example we can store 10^{21} inside a string `num1 = "1,000,000,000,000,000,000,000"` without any problem whereas this is already overflow in a 64-bit C/C++ `unsigned long long` (or Java `long`). Then, for common mathematical operations, we can use a kind of digit by digit operations to process the two BigInteger operands. For example with `num2 = "173"`, we have `num1 + num2` as:

```
num1           =    1,000,000,000,000,000,000,000
num2           =                              173
               ------------------------------- +
num1 + num2 =    1,000,000,000,000,000,000,173
```

We can also compute `num1 * num2` as:

```
num1           =    1,000,000,000,000,000,000,000
num2           =                              173
                 ------------------------------- *
                   3,000,000,000,000,000,000,000
                  70,000,000,000,000,000,000,00
                 100,000,000,000,000,000,000,0
                 ------------------------------- +
num1 * num2 = 173,000,000,000,000,000,000,000
```

Addition and subtraction are the two simpler operations in BigInteger. Multiplication takes a bit more programming job, as seen in the example above. Implementing efficient division and raising an integer to a certain power are more complicated. Anyway, coding these library routines in C/C++ under stressful contest environment can be a buggy affair, even if we can bring notes containing such C/C++ library in ICPC[2]. Fortunately, Java has a BigInteger class that we can use for this purpose. As of 24 May 2013, the C++ STL does not have such feature thus it is a good idea to use Java for BigInteger problems.

The Java BigInteger (we abbreviate it as `BI`) class supports the following basic integer operations: addition—`add(BI)`, subtraction—`subtract(BI)`, multiplication—`multiply(BI)`, power—`pow(int exponent)`, division—`divide(BI)`, remainder—`remainder(BI)`, modulo—`mod(BI)` (different to `remainder(BI)`), division and remainder—`divideAndRemainder(BI)`, and a few other interesting functions discussed later. All are just 'one liner'.

---

[1]Actually, a primitive data type also stores numbers as *limited string of bits* in computer memory. For example a 32-bit `int` data type stores an integer as 32 bits of binary string. BigInteger technique is just a generalization of this technique that uses decimal form (base 10) and longer string of digits. Note: Java BigInteger class likely uses a more efficient method than the one shown in this section.

[2]Good news for IOI contestants. IOI tasks usually do not require contestants to deal with BigInteger.

However, we need to remark that all BigInteger operations are *inherently slower* than the same operations on standard 32/64-bit integer data types. Rule of Thumb: If you can use another algorithm that only requires built-in integer data type to solve your mathematical problem, then use it instead of resorting to BigInteger.

For those who are new to Java BigInteger class, we provide the following short Java code, which is the solution for UVa 10925 - Krakovia. This problem requires BigInteger addition (to sum N large bills) and division (to divide the large sum to F friends). Observe how short and clear the code is compared to if you have to write your own BigInteger routines.

```java
import java.util.Scanner;                        // inside package java.util
import java.math.BigInteger;                      // inside package java.math

class Main {                                      /* UVa 10925 - Krakovia */
  public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    int caseNo = 1;
    while (true) {
      int N = sc.nextInt(), F = sc.nextInt();          // N bills, F friends
      if (N == 0 && F == 0) break;
      BigInteger sum = BigInteger.ZERO;      // BigInteger has this constant
      for (int i = 0; i < N; i++) {                    // sum the N large bills
        BigInteger V = sc.nextBigInteger(); // for reading next BigInteger!
        sum = sum.add(V);                            // this is BigInteger addition
      }
      System.out.println("Bill #" + (caseNo++) + " costs " + sum +
        ": each friend should pay " + sum.divide(BigInteger.valueOf(F)));
      System.out.println();          // the line above is BigInteger division
    }                                      // divide the large sum to F friends
  }
}
```

Source code: `ch5_01_UVa10925.java`

Exercise 5.3.1.1: Compute the last non zero digit of 25!; Can we use built-in data type?

Exercise 5.3.1.2: Check if 25! is divisible by 9317; Can we use built-in data type?

5.3.2 Bonus Features

Java BigInteger class has a few more bonus features that can be useful during programming contests—in terms of shortening the code length—compared to if we have to write these functions ourselves[3]. Java BigInteger class happens to have a built-in base number converter: The class's constructor and function `toString(int radix)`, a very good (but probabilistic) prime testing function `isProbablePrime(int certainty)`, a GCD routine `gcd(BI)`, and a modular arithmetic function `modPow(BI exponent, BI m)`. Among these bonus features, the base number converter is the most useful one, followed by the prime testing function. These bonus features are shown with four example problems from UVa online judge.

---

[3]A note for pure C/C++ programmers: It is good to be a *multi*-lingual programmer by switching to Java whenever it is more beneficial to do so.

Base Number Conversion

See an example below for UVa 10551 - Basic Remains. Given a base b and two non-negative integers p and m—both in base b, compute p % m and print the result as a base b integer. The base number conversion is actually a not-so-difficult[4] mathematical problem, but this problem can be made even simpler with Java BigInteger class. We can construct and print a Java BigInteger instance in any base (radix) as shown below:

```
class Main {                                      /* UVa 10551 - Basic Remains */
  public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    while (true) {
      int b = sc.nextInt();
      if (b == 0) break;                          // special class's constructor!
      BigInteger p = new BigInteger(sc.next(), b);  // the second parameter
      BigInteger m = new BigInteger(sc.next(), b);          // is the base
      System.out.println((p.mod(m)).toString(b)); // can output in any base
} } }
```

Source code: `ch5_02_UVa10551.java`

(Probabilistic) Prime Testing

Later in Section 5.5.1, we will discuss Sieve of Eratosthenes algorithm and a deterministic prime testing algorithm that is good enough for many contest problems. However, you have to type in a few lines of C/C++/Java code to do that. If you just need to check whether a single (or at most, several[5]) and usually large integer is a prime, e.g. UVa 10235 below, there is an alternative and shorter approach with function `isProbablePrime` in Java BigInteger— a probabilistic prime testing function based on Miller-Rabin's algorithm [44, 55]. There is an important parameter of this function: `certainty`. If this function returns true, then the probability that the tested BigInteger is a prime exceeds $1 - \frac{1}{2}^{certainty}$. For typical contest problems, `certainty = 10` should be enough as $1 - (\frac{1}{2})^{10} = 0.9990234375$ is ≈ 1.0. Note that using larger value of `certainty` obviously decreases the probability of WA but doing so slows down your program and thus increases the risk of TLE. Please attempt **Exercise 5.3.2.3*** to convince yourself.

```
class Main {                                      /* UVa 10235 - Simply Emirp */
  public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    while (sc.hasNext()) {
      int N = sc.nextInt();
      BigInteger BN = BigInteger.valueOf(N);
      String R = new StringBuffer(BN.toString()).reverse().toString();
      int RN = Integer.parseInt(R);
```

---

[4]For example, to convert 132 in base 8 (octal) into base 2 (binary), we can use base 10 (decimal) as the intermediate step: $(132)_8$ is $1 \times 8^2 + 3 \times 8^1 + 2 \times 8^0 = 64 + 24 + 2 = (90)_{10}$ and $(90)_{10}$ is $90 \to 45(0) \to 22(1) \to 11(0) \to 5(1) \to 2(1) \to 1(0) \to 0(1) = (1011010)_2$ (that is, divide by 2 until 0, then read the remainders from backwards).

[5]Note that if your aim is to generate a list of the first few million prime numbers, the Sieve of Eratosthenes algorithm shown in Section 5.5.1 should run faster than a few million calls of this: `isProbablePrime` function.

```
      BigInteger BRN = BigInteger.valueOf(RN);
      System.out.printf("%d is ", N);
      if (!BN.isProbablePrime(10)) // certainty 10 is enough for most cases
        System.out.println("not prime.");
      else if (N != RN && BRN.isProbablePrime(10))
        System.out.println("emirp.");
      else
        System.out.println("prime.");
} } }
```

Source code: `ch5_03_UVa10235.java`

Greatest Common Divisor (GCD)

See an example below for UVa 10814 - Simplifying Fractions. We are asked to reduce a large fraction to its simplest form by dividing both numerator and denominator with their GCD. Also see Section 5.5.2 for more details about GCD.

```
class Main {                        /* UVa 10814 - Simplifying Fractions */
  public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    int N = sc.nextInt();
    while (N-- > 0) {   // unlike in C/C++, we have to use > 0 in (N-- > 0)
      BigInteger p = sc.nextBigInteger();
      String ch = sc.next();        // we ignore the division sign in input
      BigInteger q = sc.nextBigInteger();
      BigInteger gcd_pq = p.gcd(q);                              // wow :)
      System.out.println(p.divide(gcd_pq) + " / " + q.divide(gcd_pq));
} } }
```

Source code: `ch5_04_UVa10814.java`

Modulo Arithmetic

See an example below for UVa 1230 (LA 4104) - MODEX that computes $x^y \pmod n$. Also see Section 5.5.8 and 9.21 to see how this `modPow` function is actually computed.

```
class Main {                            /* UVa 1230 (LA 4104) - MODEX */
  public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    int c = sc.nextInt();
    while (c-- > 0) {
      BigInteger x = BigInteger.valueOf(sc.nextInt());  // valueOf converts
      BigInteger y = BigInteger.valueOf(sc.nextInt());    // simple integer
      BigInteger n = BigInteger.valueOf(sc.nextInt());   // into BigInteger
      System.out.println(x.modPow(y, n));                // it's in the library!
} } }
```

Source code: `ch5_05_UVa1230.java`

Exercise 5.3.2.1: Try solving UVa 389 using the Java BigInteger technique presented here. Can you pass the time limit? If no, is there a (slightly) better technique?

Exercise 5.3.2.2*: As of 24 May 2013, programming contest problems involving *arbitrary precision* decimal numbers (not necessarily integers) are still rare. So far, we have only identified two problems in UVa online judge that require such feature: UVa 10464 and UVa 11821. Try solving these two problems using another library: Java `BigDecimal` class! Explore: `http://docs.oracle.com/javase/7/docs/api/java/math/BigDecimal.html`.

Exercise 5.3.2.3*: Write a Java program to *empirically* determine the lowest value of parameter `certainty` so that our program can run fast and there is no *composite number* between [2..10M]—a typical contest problem range—is accidentally reported as prime by `isProbablePrime(certainty)`! As `isProbablePrime` uses a probabilistic algorithm, you have to repeat your experiment several times for each `certainty` value. Is `certainty = 5` good enough? What about `certainty = 10`? What about `certainty = 1000`?

Exercise 5.3.2.4*: Study and implement the Miller Rabin's algorithm (see [44, 55]) in case you have to implement it in C/C++!

Programming Exercises related to BigInteger **NOT**[6] mentioned elsewhere:

- Basic Features
    1. UVa 00424 - Integer Inquiry (BigInteger addition)
    2. UVa 00465 - Overflow (BigInteger add/multiply, compare with $2^{31} - 1$)
    3. UVa 00619 - Numerically Speaking (BigInteger)
    4. **UVa 00713 - Adding Reversed ... *** (BigInteger + StringBuffer reverse())
    5. UVa 00748 - Exponentiation (BigInteger exponentiation)
    6. UVa 01226 - Numerical surprises (LA 3997, Danang07, mod operation)
    7. UVa 10013 - Super long sums (BigInteger addition)
    8. UVa 10083 - Division (BigInteger + number theory)
    9. UVa 10106 - Product (BigInteger multiplication)
    10. UVa 10198 - Counting (recurrences, BigInteger)
    11. *UVa 10430 - Dear GOD* (BigInteger, derive formula first)
    12. *UVa 10433 - Automorphic Numbers* (BigInteger, pow, substract, mod)
    13. UVa 10494 - If We Were a Child Again (BigInteger division)
    14. UVa 10519 - Really Strange (recurrences, BigInteger)
    15. **UVa 10523 - Very Easy *** (BigInteger addition, multiplication, and power)
    16. UVa 10669 - Three powers (BigInteger is for 3^n, binary rep of set!)
    17. UVa 10925 - Krakovia (BigInteger addition and division)
    18. *UVa 10992 - The Ghost of Programmers* (input size is up to 50 digits)
    19. UVa 11448 - Who said crisis? (BigInteger subtraction)
    20. *UVa 11664 - Langton's Ant* (simple simulation involving BigInteger)
    21. UVa 11830 - Contract revision (use BigInteger string representation)
    22. **UVa 11879 - Multiple of 17 *** (BigInteger mod, divide, subtract, equals)
    23. *UVa 12143 - Stopping Doom's Day* (LA 4209, Dhaka08, formula simplification—the hard part; use BigInteger—the easy part)
    24. *UVa 12459 - Bees' ancestors* (draw the ancestor tree to see the pattern)

---

[6]It worth mentioning that there are *many* other programming exercises in other sections of this chapter (and also in another chapters) that also use BigInteger technique.

- Bonus Feature: Base Number Conversion

  1. UVa 00290 - Palindroms ←→ ... (also involving palindrome)
  2. **UVa 00343 - What Base Is This? *** (try all possible pair of bases)
  3. UVa 00355 - The Bases Are Loaded (basic base number conversion)
  4. **UVa 00389 - Basically Speaking *** (use Java `Integer` class)
  5. UVa 00446 - Kibbles 'n' Bits 'n' Bits ... (base number conversion)
  6. UVa 10473 - Simple Base Conversion (Decimal to Hexadecimal and vice versa; if you use C/C++, you can use `strtol`)
  7. **UVa 10551 - Basic Remains *** (also involving BigInteger mod)
  8. UVa 11185 - Ternary (Decimal to base 3)
  9. *UVa 11952 - Arithmetic* (check base 2 to 18 only; special case for base 1)

- Bonus Feature: Primality Testing

  1. *UVa 00960 - Gaussian Primes* (there is a number theory behind this)
  2. **UVa 01210 - Sum of Consecutive ... *** (LA 3399, Tokyo05, simple)
  3. **UVa 10235 - Simply Emirp *** (case analysis: not prime/prime/emirp; emirp is defined as prime number that if reversed is still a prime number)
  4. UVa 10924 - Prime Words (check if sum of letter values is a prime)
  5. **UVa 11287 - Pseudoprime Numbers *** (output yes if `!isPrime(p) + a.modPow(p, p) = a`; use Java BigInteger)
  6. *UVa 12542 - Prime Substring* (HatYai12, brute force, use isProbablePrime to test primality)

- Bonus Feature: Others

  1. **UVa 01230 - MODEX *** (LA 4104, Singapore07, BigInteger modPow)
  2. *UVa 10023 - Square root* (code Newton's method with BigInteger)
  3. UVa 10193 - All You Need Is Love (convert two binary strings S1 and S2 to decimal and check see if gcd(s1, s2) > 1)
  4. UVa 10464 - Big Big Real Numbers (solvable with Java BigDecimal class)
  5. **UVa 10814 - Simplifying Fractions *** (BigInteger gcd)
  6. **UVa 11821 - High-Precision Number *** (Java BigDecimal class)

Profile of Algorithm Inventors

Gary Lee Miller is a professor of Computer Science at Carnegie Mellon University. He is the initial inventor of Miller-Rabin primality test algorithm.

Michael Oser Rabin (born 1931) is an Israeli computer scientist. He improved Miller's idea and invented the Miller-Rabin primality test algorithm. Together with Richard Manning Karp, he also invented Rabin-Karp's string matching algorithm.

5.4 Combinatorics

Combinatorics is a branch of *discrete mathematics*[7] concerning the study of **countable** discrete structures. In programming contests, problems involving combinatorics are usually titled 'How Many [Object]', 'Count [Object]', etc, although some problem authors choose to hide this fact from their problem titles. The solution code is usually *short*, but finding the (usually recursive) formula takes some mathematical brilliance and also patience.

In ICPC[8], if such a problem exists in the given problem set, ask one team member who is strong in mathematics to derive the formula whereas the other two concentrate on *other* problems. Quickly code the usually short formula once it is obtained—interrupting whoever is currently using the computer. It is also a good idea to memorize/study the common ones like the Fibonacci-related formulas (see Section 5.4.1), Binomial Coefficients (see Section 5.4.2), and Catalan Numbers (see Section 5.4.3).

Some of these combinatorics formulas may yield overlapping subproblems that entails the need of using Dynamic Programming technique (see Section 3.5). Some computation values can also be large that entails the need of using BigInteger technique (see Section 5.3).

5.4.1 Fibonacci Numbers

Leonardo *Fibonacci*'s numbers are defined as $fib(0) = 0$, $fib(1) = 1$, and for $n \geq 2$, $fib(n) = fib(n - 1) + fib(n - 2)$. This generates the following familiar pattern: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, and so on. This pattern sometimes appears in contest problems which do not mention the term 'Fibonacci' at all, like in some problems in the list of programming exercises in this section (e.g. UVa 900, 10334, 10450, 10497, 10862, etc).

We usually derive the Fibonacci numbers with a 'trivial' $O(n)$ DP technique and not implement the given recurrence directly (as it is very slow). However, the $O(n)$ DP solution is *not* the fastest for all cases. Later in Section 9.21, we will show how to compute the n-th Fibonacci number (where n is large) in $O(\log n)$ time using the efficient matrix power. As a note, there is an $O(1)$ *approximation* technique to get the *n-th* Fibonacci number. We can compute the closest integer of $(\phi^n - (-\phi)^{-n})/\sqrt{5}$ (Binet's formula) where ϕ (golden ratio) is $((1 + \sqrt{5})/2) \approx 1.618$. However this is not so accurate for large Fibonacci numbers.

Fibonacci numbers grow very fast and some problems involving Fibonacci have to be solved using Java BigInteger library (see Section 5.3).

Fibonacci numbers have many interesting properties. One of them is the **Zeckendorf's theorem**: Every positive integer can be written in a unique way as a sum of one or more distinct Fibonacci numbers such that the sum does not include any two consecutive Fibonacci numbers. For any given positive integer, a representation that satisfies Zeckendorf's theorem can be found by using a *Greedy* algorithm: Choose the largest possible Fibonacci number at each step. For example: 100 = 89 + 8 + 3; 77 = 55 + 21 + 1, 18 = 13 + 5, etc.

Another property is the **Pisano Period** where the last one/last two/last three/last four digit(s) of a Fibonacci number repeats with a period of 60/300/1500/15000, respectively.

Exercise 5.4.1.1: Try $fib(n) = (\phi^n - (-\phi)^{-n})/\sqrt{5}$ on small n and see if this Binet's formula really produces $fib(7) = 13$, $fib(9) = 34$, $fib(11) = 89$. Now, write a simple program to find out the first value of n such that the actual value of $fib(n)$ differs from the result of this approximation formula? Is that n big enough for typical usage in programming contests?

---

[7]Discrete mathematics is a study of structures that are discrete (e.g. integers {0, 1, 2, ... }, graphs/trees (vertices and edges), logic (true/false)) rather than continuous (e.g. real numbers).
[8]Note that pure combinatorics problem is rarely used in an IOI task (it can be part of a bigger task).

5.4.2 Binomial Coefficients

Another classical combinatorics problem is in finding the *coefficients* of the algebraic expansion of powers of a binomial[9]. These coefficients are also the number of ways that n items can be taken k at a time, usually written as $C(n, k)$ or ^nC_k. For example, $(x + y)^3 = \mathbf{1}x^3 + \mathbf{3}x^2y + \mathbf{3}xy^2 + \mathbf{1}y^3$. The $\{\mathbf{1, 3, 3, 1}\}$ are the binomial coefficients of $n = 3$ with $k = \{0, 1, 2, 3\}$ respectively. Or in other words, the number of ways that $n = 3$ items can be taken $k = \{0, 1, 2, 3\}$ item at a time are $\{1, 3, 3, 1\}$ different ways, respectively.

We can compute a single value of $C(n, k)$ with this formula: $C(n, k) = \frac{n!}{(n-k)! \times k!}$. However, computing $C(n, k)$ can be a challenge when n and/or k are large. There are several tricks like: Making k smaller (if $k > n - k$, then we set $k = n - k$ because $^nC_k =^n C_{(n-k)}$; during intermediate computations, we divide the numbers first before multiply it with the next number; or use BigInteger technique (last resort as BigInteger operations are slow).

If we have to compute *many but not all* values of $C(n, k)$ for different n and k, it is better to use top-down Dynamic Programming. We can write $C(n, k)$ as shown below and use a 2D memo table to avoid re-computations.

$C(n, 0) = C(n, n) = 1$ // base cases.
$C(n, k) = C(n - 1, k - 1) + C(n - 1, k)$ // take or ignore an item, $n > k > 0$.

However, if we have to compute *all* values of $C(n, k)$ from $n = 0$ up to a certain value of n, then it may be beneficial to construct the *Pascal's Triangle*, a triangular array of binomial coefficients. The leftmost and rightmost entries at each row are always 1. The inner values are the sum of two values directly above it, as shown for row $n = 4$ below. This is essentially the bottom-up version of the DP solution above.

```
n = 0                1
n = 1              1   1
n = 2            1   2   1
n = 3          1   3   3   1   <- as shown above
                \ / \ / \ /
n = 4      1    4   6   4    1 ... and so on
```

Exercise 5.4.2.1: A frequently used k for $C(n, k)$ is $k = 2$. Show that $C(n, 2) = O(n^2)$.

5.4.3 Catalan Numbers

First, let's define the n-th Catalan number—written using binomial coefficients notation ^nC_k above—as: $Cat(n) = (^{(2 \times n)}C_n)/(n + 1)$; $Cat(0) = 1$. We will see its purpose below.

If we are asked to compute the values of $Cat(n)$ for *several* values of n, it may be better to compute the values using bottom-up Dynamic Programming. If we know $Cat(n)$, we can compute $Cat(n + 1)$ by manipulating the formula like shown below.

$Cat(n) = \frac{2n!}{n! \times n! \times (n+1)}$.

$Cat(n + 1) = \frac{(2 \times (n+1))!}{(n+1)! \times (n+1)! \times ((n+1)+1)} = \frac{(2n+2) \times (2n+1) \times 2n!}{(n+1) \times n! \times (n+1) \times n! \times (n+2)} = \frac{(2n+2) \times (2n+1) \times ... [2n!]}{(n+2) \times (n+1) \times ... [n! \times n! \times (n+1)]}$.

Therefore, $Cat(n + 1) = \frac{(2n+2) \times (2n+1)}{(n+2) \times (n+1)} \times Cat(n)$.

Alternatively, we can set $m = n + 1$ so that we have: $Cat(m) = \frac{2m \times (2m-1)}{(m+1) \times m} \times Cat(m - 1)$.

---
[9]Binomial is a special case of polynomial that only has two terms.

Catalan numbers are found in various combinatorial problems. Here, we list down some of the more interesting ones (there are several others, see **Exercise 5.4.4.8***). All examples below use $n = 3$ and $Cat(3) = ({}^{(2 \times 3)}C_3)/(3 + 1) = ({}^{6}C_3)/4 = 20/4 = 5$.

1. $Cat(n)$ counts the number of distinct binary trees with n vertices, e.g. for $n = 3$:

```
   *      *      *      *      *
  /      /      / \      \      \
 *      *      *   *      *      *
 /      \              /          \
*        *            *            *
```

2. $Cat(n)$ counts the number of expressions containing n pairs of parentheses which are correctly matched, e.g. for $n = 3$, we have: ()()(), ()(()), (())(), ((())), and (()()).

3. $Cat(n)$ counts the number of different ways $n + 1$ factors can be completely parenthesized, e.g. for $n = 3$ and $3 + 1 = 4$ factors: {a, b, c, d}, we have: (ab)(cd), a(b(cd)), ((ab)c)d, (a(bc))(d), and a((bc)d).

4. $Cat(n)$ counts the number of ways a convex polygon (see Section 7.3) of $n + 2$ sides can be triangulated. See Figure 5.1, left.

5. $Cat(n)$ counts the number of monotonic paths along the edges of an $n \times n$ grid, which do not pass above the diagonal. A monotonic path is one which starts in the lower left corner, finishes in the upper right corner, and consists entirely of edges pointing rightwards or upwards. See Figure 5.1, right and also see Section 4.7.1.
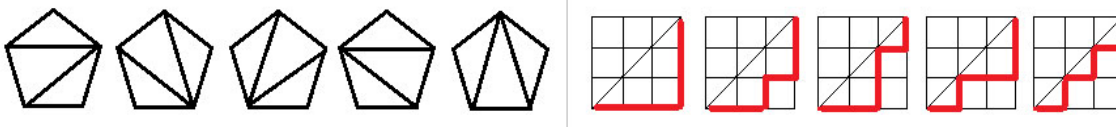
Figure 5.1: Left: Triangulation of a Convex Polygon, Right: Monotonic Paths

5.4.4 Remarks about Combinatorics in Programming Contests

There are many other combinatorial problems that may also appear in programming contests, but they are not as frequent as Fibonacci numbers, Binomial Coefficients, or Catalan numbers. Some of the more interesting ones are listed in Section 9.8.

In *online* programming contests where contestant can access the Internet, there is one more trick that may be useful. First, generate the output for small instances and then search for that sequence at OEIS (The On-Line Encyclopedia of Integer Sequences) hosted at http://oeis.org/. If you are lucky, OEIS can tell you the name of the sequence and/or the required general formula for the larger instances.

There are still many other counting principles and formulas, too many to be discussed in this book. We close this section by giving some written exercises to test/further improve your combinatorics skills. Note: The problems listed in this section constitute $\approx 15\%$ of the entire problems in this chapter.

Exercise 5.4.4.1: Count the number of different possible outcomes if you roll two 6-sided dices and flip two 2-sided coins?

Exercise 5.4.4.2: How many ways to form a three digits number from $\{0, 1, 2, \ldots, 9\}$ and each digit can only be used once? Note that 0 cannot be used as the leading digit.

Exercise 5.4.4.3: Suppose you have a 6-letters word 'FACTOR'. If we take 3-letters from this word 'FACTOR', we may have another valid English word, like 'ACT', 'CAT', 'ROT', etc. What is the maximum number of different 3-letters word that can be formed with the letters from 'FACTOR'? You do not have to care whether the 3-letters word is a valid English word or not.

Exercise 5.4.4.4: Suppose you have a 5-letters word 'BOBBY'. If we rearrange the letters, we can get another word, like 'BBBOY', 'YOBBB', etc. How many *different* permutations are possible?

Exercise 5.4.4.5: Solve UVa 11401 - Triangle Counting! This problem has a short description: "Given n rods of length 1, 2, ..., n, pick any 3 of them and build a triangle. How many distinct triangles can you make (consider triangle inequality, see Section 7.2)? $(3 \leq n \leq 1M)$". Note that, two triangles will be considered different if they have at least one pair of arms with different lengths. If you are lucky, you may spend only a few minutes to spot the pattern. Otherwise, this problem may end up unsolved by the time contest is over—which is a bad sign for your team.

Exercise 5.4.4.6*: Study the following terms: Burnside's Lemma, Stirling Numbers.

Exercise 5.4.4.7*: Which one is the hardest to factorize (see Section 5.5.4) assuming that n is an arbitrary large integer: $fib(n)$, $C(n, k)$ (assume that $k = n/2$), or $Cat(n)$? Why?

Exercise 5.4.4.8*: Catalan numbers $Cat(n)$ appear in some other interesting problems other than the ones shown in this section. Investigate!

Other Programming Exercises related to Combinatorics:

- Fibonacci Numbers
    1. UVa 00495 - Fibonacci Freeze (very easy with Java BigInteger)
    2. UVa 00580 - Critical Mass (related to *Tribonacci* series; Tribonacci numbers are the generalization of Fibonacci numbers; it is defined by $T_1 = 1$, $T_2 = 1$, $T_3 = 2$, and $T_n = T_{n-1} + T_{n-2} + T_{n-3}$ for $n \geq 4$)
    3. **UVa 00763 - Fibinary Numbers *** (Zeckendorf representation, greedy, use Java BigInteger)
    4. UVa 00900 - Brick Wall Patterns (combinatorics, the pattern $\approx$ Fibonacci)
    5. UVa 00948 - Fibonaccimal Base (Zeckendorf representation, greedy)
    6. UVa 01258 - Nowhere Money (LA 4721, Phuket09, Fibonacci variant, Zeckendorf representation, greedy)
    7. UVa 10183 - How many Fibs? (get the number of Fibonaccis when generating them; BigInteger)
    8. **UVa 10334 - Ray Through Glasses *** (combinatorics, Java BigInteger)
    9. UVa 10450 - World Cup Noise (combinatorics, the pattern $\approx$ Fibonacci)
    10. UVa 10497 - Sweet Child Make Trouble (the pattern $\approx$ Fibonacci)

11. UVa 10579 - Fibonacci Numbers (very easy with Java BigInteger)
12. **UVa 10689 - Yet Another Number ... *** (easy if you know Pisano (a.k.a Fibonacci) period)
13. UVa 10862 - Connect the Cable Wires (the pattern ends up $\approx$ Fibonacci)
14. UVa 11000 - Bee (combinatorics, the pattern is similar to Fibonacci)
15. *UVa 11089 - Fi-binary Number* (the list of Fi-binary Numbers follow the Zeckendorf's theorem)
16. UVa 11161 - Help My Brother (II) (Fibonacci + median)
17. UVa 11780 - Miles 2 Km (the background problem is Fibonacci numbers)

- Binomial Coefficients:

  1. UVa 00326 - Extrapolation using a ... (difference table)
  2. UVa 00369 - Combinations (be careful with overflow issue)
  3. UVa 00485 - Pascal Triangle of Death (binomial coefficients + BigInteger)
  4. UVa 00530 - Binomial Showdown (work with doubles; optimize computation)
  5. *UVa 00911 - Multinomial Coefficients* (there is a formula for this, $result = n!/(z_1! \times z_2! \times z_3! \times ... \times zk!)$)
  6. UVa 10105 - Polynomial Coefficients ($n!/(n_1! \times n_2! \times ... \times n_k!)$); however, the derivation is complex)
  7. **UVa 10219 - Find the Ways *** (count the length of ${}^nC_k$; BigInteger)
  8. UVa 10375 - Choose and Divide (the main task is to avoid overflow)
  9. *UVa 10532 - Combination, Once Again* (modified binomial coefficient)
  10. **UVa 10541 - Stripe *** (a good combinatorics problem; compute how many white cells are there via $Nwhite = N$ - sum of all K integers; imagine we have one more white cell at the very front, we can now determine the answer by placing black stripes after K out of $Nwhite + 1$ whites, or ${}_{Nwhite+1}C_K$ (use Java BigInteger); however, if $K > Nwhite + 1$ then the answer is 0)
  11. **UVa 11955 - Binomial Theorem *** (pure application; DP)

- Catalan Numbers

  1. **UVa 00991 - Safe Salutations *** (Catalan Numbers)
  2. **UVa 10007 - Count the Trees *** (answer is $Cat(n) \times n!$; BigInteger)
  3. UVa 10223 - How Many Nodes? (Precalculate the answers as there are only 19 Catalan Numbers $< 2^{32} - 1$)
  4. UVa 10303 - How Many Trees (generate $Cat(n)$ as shown in this section, use Java BigInteger)
  5. **UVa 10312 - Expression Bracketing *** (the number of binary bracketing can be counted by $Cat(n)$; the total number of bracketing can be computed using *Super-Catalan* numbers)
  6. *UVa 10643 - Facing Problems With ...* ($Cat(n)$ is part of a bigger problem)

- Others, Easier

  1. UVa 11115 - Uncle Jack (N^D, use Java BigInteger)
  2. **UVa 11310 - Delivery Debacle *** (requires DP: let `dp[i]` be the number of ways the cakes can be packed for a box $2 \times$ `i`. Note that it is possible to use two 'L shaped' cakes to form a 2×3 shape)
  3. **UVa 11401 - Triangle Counting *** (spot the pattern, coding is easy)
  4. UVa 11480 - Jimmy's Balls (try all r, but simpler formula exists)
  5. **UVa 11597 - Spanning Subtree *** (uses knowledge of graph theory, the answer is very trivial)

6. UVa 11609 - Teams ($N \times 2^{N-1}$, use Java BigInteger for the modPow part)

7. *UVa 12463 - Little Nephew* (double socks & shoes to simplify the problem)

- Others, Harder

  1. *UVa 01224 - Tile Code* (derive formula from small instances)
  2. UVa 10079 - Pizza Cutting (derive the one liner formula)
  3. UVa 10359 - Tiling (derive the formula, use Java BigInteger)
  4. UVa 10733 - The Colored Cubes (Burnside's lemma)
  5. **UVa 10784 - Diagonal *** (the number of diagonals in n-gon $= n*(n-3)/2$, use it to derive the solution)
  6. UVa 10790 - How Many Points of ... (uses arithmetic progression formula)
  7. UVa 10918 - Tri Tiling (there are two related recurrences here)
  8. **UVa 11069 - A Graph Problem *** (use Dynamic Programming)
  9. UVa 11204 - Musical Instruments (only first choice matters)
  10. *UVa 11270 - Tiling Dominoes* (sequence A004003 in OEIS)
  11. ***UVa 11538 - Chess Queen **** (count along rows, columns, and diagonals)
  12. UVa 11554 - Hapless Hedonism (similar to UVa 11401)
  13. *UVa 12022 - Ordering T-shirts* (number of ways n competitors can rank in a competition, allowing for the possibility of ties, see `http://oeis.org/A000670`)

Profile of Algorithm Inventors

Leonardo Fibonacci (also known as **Leonardo Pisano**) (1170-1250) was an Italian mathematician. He published a book titled 'Liber Abaci' (Book of Abacus/Calculation) in which he discussed a problem involving the growth of a population of *rabbits* based on idealized assumptions. The solution was a sequence of numbers now known as the Fibonacci numbers.

Edouard Zeckendorf (1901-1983) was a Belgian mathematician. He is best known for his work on Fibonacci numbers and in particular for proving Zeckendorf's theorem.

Jacques Philippe Marie Binet (1786-1856) was a French mathematician. He made significant contributions to number theory. Binet's formula expressing Fibonacci numbers in closed form is named in his honor, although the same result was known earlier.

Blaise Pascal (1623-1662) was a French mathematician. One of his famous invention discussed in this book is the Pascal's triangle of binomial coefficients.

Eugène Charles Catalan (1814-1894) was a French and Belgian mathematician. He is the one who introduced the Catalan numbers to solve a combinatorial problem.

Eratosthenes of Cyrene ($\approx$ 300-200 years BC) was a Greek mathematician. He invented geography, did measurements of the circumference of earth, and invented a simple algorithm to generate prime numbers which we discuss in this book.

Leonhard Euler (1707-1783) was a Swiss mathematician. His inventions mentioned in this book are the Euler totient (Phi) function and the Euler tour/path (Graph).

Christian Goldbach (1690-1764) was a German mathematician. He is remembered today for Goldbach's conjecture that he discussed extensively with Leonhard Euler.

Diophantus of Alexandria ($\approx$ 200-300 AD) was an Alexandrian Greek mathematician. He did a lot of study in algebra. One of his works is the Linear Diophantine Equations.

5.5 Number Theory

Mastering as many topics as possible in the field of *number theory* is important as some mathematics problems become easy (or easier) if you know the theory behind the problems. Otherwise, either a plain brute force attack leads to a TLE response or you simply cannot work with the given input as it is too large without some pre-processing.

5.5.1 Prime Numbers

A natural number starting from 2: $\{2, 3, 4, 5, 6, 7, \ldots\}$ is considered as a **prime** if it is only divisible by 1 or itself. The first and the only even prime is 2. The next prime numbers are: 3, 5, 7, 11, 13, 17, 19, 23, 29, ..., and infinitely many more primes (proof in [56]). There are 25 primes in range `[0..100]`, 168 primes in `[0..1000]`, 1000 primes in `[0..7919]`, 1229 primes in `[0..10000]`, etc. Some large prime numbers are[10] 104729, 1299709, 15485863, 179424673, 2147483647, 32416190071, 112272535095293, 48112959837082048697, etc.

Prime number is an important topic in number theory and the source for many programming problems[11]. In this section, we will discuss algorithms involving prime numbers.

Optimized Prime Testing Function

The first algorithm presented in this section is for testing whether a given natural number N is prime, i.e. `bool isPrime(N)`. The most naïve version is to test by definition, i.e. test if N is divisible by *divisor* $\in$ `[2..N-1]`. This works, but runs in $O(N)$—in terms of number of divisions. This is not the best way and there are several possible improvements.

The first major improvement is to test if N is divisible by a *divisor* $\in$ `[2..`$\sqrt{N}$`]`, i.e. we stop when the *divisor* is greater than $\sqrt{N}$. Reason: If N is divisible by d, then $N = d \times \frac{N}{d}$. If $\frac{N}{d}$ is smaller than d, then $\frac{N}{d}$ or a prime factor of $\frac{N}{d}$ would have divided N earlier. Therefore d and $\frac{N}{d}$ cannot *both* be greater than $\sqrt{N}$. This improvement is $O(\sqrt{N})$ which is already much faster than the previous version, but can still be improved to be twice as fast.

The second improvement is to test if N is divisible by *divisor* $\in$ `[3, 5, 7,..,` $\sqrt{N}$`]`, i.e. we only test odd numbers up to $\sqrt{N}$. This is because there is only one even prime number, i.e. number 2, which can be tested separately. This is $O(\sqrt{N}/2)$, which is also $O(\sqrt{N})$.

The third improvement[12] which is already good enough[13] for contest problems is to test if N is divisible by *prime divisors* $\leq \sqrt{N}$. This is because if a prime number X cannot divide N, then there is no point testing whether multiples of X divide N or not. This is faster than $O(\sqrt{N})$ which is about $O(\#primes \leq \sqrt{N})$. For example, there are 500 odd numbers in `[1..`$\sqrt{10^6}$`]`, but there are only 168 primes in the same range. Prime number theorem [56] says that the number of primes less than or equal to M—denoted by $\pi(M)$—is bounded by $O(M/(\ln(M) - 1))$. Therefore, the complexity of this prime testing function is about $O(\sqrt{N}/\ln(\sqrt{N}))$. The code is shown in the next discussion below.

Sieve of Eratosthenes: Generating List of Prime Numbers

If we want to generate a list of prime numbers between range `[0..N]`, there is a better algorithm than testing each number in the range whether it is a prime number or not. The

---

[10]Having a list of large random prime numbers can be good for testing as these are the numbers that are hard for algorithms like the prime testing or prime factoring algorithms.

[11]In real life, large primes are used in cryptography because it is hard to factor a number xy into $x \times y$ when both are **relatively prime** (also known as **coprime**).

[12]This is a bit recursive—testing whether a number is a prime by using another (smaller) prime number. But the reason should be obvious after reading the next section.

[13]Also see Section 5.3.2 for the Miller-Rabin's probabilistic prime testing with Java BigInteger class.

algorithm is called 'Sieve of *Eratosthenes*' invented by Eratosthenes of Alexandria.

First, this Sieve algorithm sets all numbers in the range to be 'probably prime' but set numbers 0 and 1 to be not prime. Then, it takes 2 as prime and crosses out all multiples[14] of 2 starting from $2 \times 2 = 4$, 6, 8, 10, ... until the multiple is greater than N. Then it takes the next non-crossed number 3 as a prime and crosses out all multiples of 3 starting from $3 \times 3 = 9$, 12, 15, ... Then it takes 5 and crosses out all multiples of 5 starting from $5 \times 5 = 25$, 30, 35, ... And so on .... After that, whatever left uncrossed within the range $[0..N]$ are primes. This algorithm does approximately $(N \times (1/2 + 1/3 + 1/5 + 1/7 + \ldots + 1/\text{last}$ prime in range $\leq N))$ operations. Using 'sum of reciprocals of primes up to n', we end up with the time complexity of *roughly* $O(N \log \log N)$.

Since generating a list of primes $\leq 10K$ using the sieve is fast (our code below can go up to 10^7 under contest setting), we opt to use sieve for smaller primes and reserve optimized prime testing function for larger primes—see previous discussion. The code is as follows:

```
#include <bitset>          // compact STL for Sieve, better than vector<bool>!
ll _sieve_size;                   // ll is defined as: typedef long long ll;
bitset<10000010> bs;              // 10^7 should be enough for most cases
vi primes;                  // compact list of primes in form of vector<int>

void sieve(ll upperbound) {     // create list of primes in [0..upperbound]
  _sieve_size = upperbound + 1;            // add 1 to include upperbound
  bs.set();                                      // set all bits to 1
  bs[0] = bs[1] = 0;                          // except index 0 and 1
  for (ll i = 2; i <= _sieve_size; i++) if (bs[i]) {
    // cross out multiples of i starting from i * i!
    for (ll j = i * i; j <= _sieve_size; j += i) bs[j] = 0;
    primes.push_back((int)i);       // add this prime to the list of primes
} }                                   // call this method in main method

bool isPrime(ll N) {             // a good enough deterministic prime tester
  if (N <= _sieve_size) return bs[N];           // O(1) for small primes
  for (int i = 0; i < (int)primes.size(); i++)
    if (N % primes[i] == 0) return false;
  return true;                // it takes longer time if N is a large prime!
}               // note: only work for N <= (last prime in vi "primes")^2

// inside int main()
  sieve(10000000);                    // can go up to 10^7 (need few seconds)
  printf("%d\n", isPrime(2147483647));              // 10-digits prime
  printf("%d\n", isPrime(136117223861LL));   // not a prime, 104729*1299709
```

Source code: ch5_06_primes.cpp/java

5.5.2 Greatest Common Divisor & Least Common Multiple

The Greatest Common Divisor (GCD) of two integers: a, b denoted by $gcd(a, b)$, is the largest positive integer d such that $d \mid a$ and $d \mid b$ where $x \mid y$ means that x divides y. Example of GCD: $gcd(4, 8) = 4$, $gcd(6, 9) = 3$, $gcd(20, 12) = 4$. One practical usage of GCD is to simplify fractions (see UVa 10814 in Section 5.3.2), e.g. $\frac{6}{9} = \frac{6/gcd(6,9)}{9/gcd(6,9)} = \frac{6/3}{9/3} = \frac{2}{3}$.

---

[14]Common implementation is to start from $2 \times$ i instead of i $\times$ i, but the difference is not that much.

Finding the GCD of two integers is an easy task with an effective Divide and Conquer *Euclid* algorithm [56, 7] which can be implemented as a one liner code (see below). Thus finding the GCD of two integers is usually not the main issue in a Math-related contest problem, but just part of a bigger solution.

The GCD is closely related to Least (or Lowest) Common Multiple (LCM). The LCM of two integers (a, b) denoted by $lcm(a, b)$, is defined as the smallest positive integer l such that $a \mid l$ and $b \mid l$. Example of LCM: $lcm(4, 8) = 8$, $lcm(6, 9) = 18$, $lcm(20, 12) = 60$. It has been shown (see [56]) that: $lcm(a, b) = a \times b/gcd(a, b)$. This can also be implemented as a one liner code (see below).

```
int gcd(int a, int b) { return b == 0 ? a : gcd(b, a % b); }
int lcm(int a, int b) { return a * (b / gcd(a, b)); }
```

The GCD of more than 2 numbers, e.g. $gcd(a, b, c)$ is equal to $gcd(a, gcd(b, c))$, etc, and similarly for LCM. Both GCD and LCM algorithms run in $O(\log_{10} n)$, where $n = max(a, b)$.

Exercise 5.5.2.1: The formula for LCM is $lcm(a, b) = a \times b/gcd(a, b)$ but why do we use $a \times (b/gcd(a, b))$ instead? Hint: Try $a = 1000000000$ and $b = 8$ using 32-bit signed integers.

5.5.3 Factorial

Factorial of n, i.e. $n!$ or $fac(n)$ is defined as 1 if $n = 0$ and $n \times fac(n - 1)$ if $n > 0$. However, it is usually more convenient to work with the iterative version, i.e. $fac(n) = 2 \times 3 \times \ldots \times (n - 1) \times n$ (loop from 2 to n, skipping 1). The value of $fac(n)$ grows very fast. We can still use C/C++ `long long` (Java `long`) for up to $fac(20)$. Beyond that, we may need to use either Java BigInteger library for precise but slow computation (see Section 5.3), work with the prime factors of a factorial (see Section 5.5.5), or get the intermediate and final results modulo a smaller number (see Section 5.5.8).

5.5.4 Finding Prime Factors with Optimized Trial Divisions

In number theory, we know that a prime number N only have 1 and itself as factors but a **composite** number N, i.e. the non-primes, can be written uniquely as a multiplication of its prime factors. That is, prime numbers are multiplicative building blocks of integers (the fundamental theorem of arithmetic). For example, $N = 1200 = 2 \times 2 \times 2 \times 2 \times 3 \times 5 \times 5 = 2^4 \times 3 \times 5^2$ (the latter form is called **prime-power factorization**).

A naïve algorithm generates a list of primes (e.g. with sieve) and check which prime(s) can actually divide the integer N—without changing N. This can be improved!

A better algorithm utilizes a kind of Divide and Conquer spirit. An integer N can be expressed as: $N = PF \times N'$, where PF is a prime factor and N' is another number which is N/PF—i.e. we can reduce the size of N by taking out its prime factor PF. We can keep doing this until eventually $N' = 1$. To speed up the process even further, we utilize the divisibility property that there is no divisor greater than $\sqrt{N}$, so we only repeat the process of finding prime factors until $PF \leq \sqrt{N}$. Stopping at $\sqrt{N}$ entails a special case: If (current $PF)^2 > N$ and N is still not 1, then N is the last prime factor. The code below takes in an integer N and returns the list of prime factors.

In the worst case—when N is prime, this prime factoring algorithm with trial division requires testing all smaller primes up to $\sqrt{N}$, mathematically denoted as $O(\pi(\sqrt{N})) = O(\sqrt{N}/ln\sqrt{N})$—see the example of factoring a large composite number 136117223861 into

two large prime factors: 104729×1299709 in the code below. However, if given composite numbers with lots of small prime factors, this algorithm is reasonably fast—see 142391208960 which is $2^{10} \times 3^4 \times 5 \times 7^4 \times 11 \times 13$.

```
vi primeFactors(ll N) {      // remember: vi is vector<int>, ll is long long
  vi factors;
  ll PF_idx = 0, PF = primes[PF_idx]; // primes has been populated by sieve
  while (PF * PF <= N) {               // stop at sqrt(N); N can get smaller
    while (N % PF == 0) { N /= PF; factors.push_back(PF); }    // remove PF
    PF = primes[++PF_idx];                      // only consider primes!
  }
  if (N != 1) factors.push_back(N);        // special case if N is a prime
  return factors;      // if N does not fit in 32-bit integer and is a prime
}                   // then 'factors' will have to be changed to vector<ll>

// inside int main(), assuming sieve(1000000) has been called before
  vi r = primeFactors(2147483647);        // slowest, 2147483647 is a prime
  for (vi::iterator i = r.begin(); i != r.end(); i++) printf("> %d\n", *i);

  r = primeFactors(136117223861LL);                   // slow, 104729*1299709
  for (vi::iterator i = r.begin(); i != r.end(); i++) printf("# %d\n", *i);

  r = primeFactors(142391208960LL);        // faster, 2^10*3^4*5*7^4*11*13
  for (vi::iterator i = r.begin(); i != r.end(); i++) printf("! %d\n", *i);
```

Exercise 5.5.4.1: Examine the given code above. What is/are the value(s) of N that can break this piece of code? You can assume that vi 'primes' contains list of prime numbers with the largest prime of 9999991 (slightly below 10 million).

Exercise 5.5.4.2: John Pollard invented a better algorithm for integer factorization. Study and implement Pollard's rho algorithm (both the original and the improvement by Richard P. Brent) [52, 3]!

5.5.5 Working with Prime Factors

Other than using the Java BigInteger technique (see Section 5.3) which is 'slow', we can work with the *intermediate computations* of large integers *accurately* by working with the *prime factors* of the integers instead of the actual integers themselves. Therefore, for some non-trivial number theoretic problems, we have to work with the prime factors of the input integers even if the main problem is not really about prime numbers. After all, prime factors are the building blocks of integers. Let's see the case study below.

  UVa 10139 - Factovisors can be abridged as follow: "Does m divides $n!$? $(0 \le n, m \le 2^{31} - 1)$". In the earlier Section 5.5.3, we mentioned that with *built-in data types*, the largest factorial that we can still compute precisely is 20!. In Section 5.3, we show that we can compute large integers with Java BigInteger technique. However, it is *very slow* to precisely compute the exact value of $n!$ for large n. The solution for this problem is to work with the prime factors of both $n!$ and m. We factorize m to its prime factors and see if it has 'support' in $n!$. For example, when $n = 6$, we have 6! expressed as prime power factorization:

$6! = 2 \times 3 \times 4 \times 5 \times 6 = 2 \times 3 \times (2^2) \times 5 \times (2 \times 3) = 2^4 \times 3^2 \times 5$. For $6!$, $m_1 = 9 = 3^2$ has support—see that 3^2 is part of $6!$, thus $m_1 = 9$ divides $6!$. However, $m_2 = 27 = 3^3$ has *no* support—see that the largest power of 3 in $6!$ is just 3^2, thus $m_2 = 27$ does *not* divide $6!$.

Exercise 5.5.5.1: Determine what is the GCD and LCM of $(2^6 \times 3^3 \times 97^1, 2^5 \times 5^2 \times 11^2)$?

5.5.6   Functions Involving Prime Factors

There are other well-known number theoretic functions involving prime factors shown below. All variants have similar time complexity with the basic prime factoring via trial division above. Interested readers can further explore Chapter 7: "Multiplicative Functions" of [56].

1. `numPF(N)`: Count the number of *prime factors* of `N`

   A simple tweak of the trial division algorithm to find prime factors shown earlier.

```
ll numPF(ll N) {
  ll PF_idx = 0, PF = primes[PF_idx], ans = 0;
  while (PF * PF <= N) {
    while (N % PF == 0) { N /= PF; ans++; }
    PF = primes[++PF_idx];
  }
  if (N != 1) ans++;
  return ans;
}
```

2. `numDiffPF(N)`: Count the number of *different* prime factors of `N`

3. `sumPF(N)`: *Sum* the prime factors of `N`

4. `numDiv(N)`: Count the number of *divisors* of `N`

   Divisor of integer N is defined as an integer that divides N without leaving a remainder. If a number $N = a^i \times b^j \times \ldots \times c^k$, then N has $(i+1) \times (j+1) \times \ldots \times (k+1)$ divisors. For example: $N = 60 = 2^2 \times 3^1 \times 5^1$ has $(2+1) \times (1+1) \times (1+1) = 3 \times 2 \times 2 = 12$ divisors. The 12 divisors are: $\{1, \mathbf{2}, \mathbf{3}, 4, \mathbf{5}, 6, 10, 12, 15, 20, 30, 60\}$. The prime factors of 12 are **highlighted**. See that N has more divisors than prime factors.

```
ll numDiv(ll N) {
  ll PF_idx = 0, PF = primes[PF_idx], ans = 1;    // start from ans = 1
  while (PF * PF <= N) {
    ll power = 0;                                 // count the power
    while (N % PF == 0) { N /= PF; power++; }
    ans *= (power + 1);                    // according to the formula
    PF = primes[++PF_idx];
  }
  if (N != 1) ans *= 2;    // (last factor has pow = 1, we add 1 to it)
  return ans;
}
```

5. `sumDiv(N)`: *Sum* the divisors of `N`

   In the previous example, $N = 60$ has 12 divisors. The sum of these divisors is 168. This can be computed via prime factors too. If a number $N = a^i \times b^j \times \ldots \times c^k$, then the sum of divisors of N is $\frac{a^{i+1}-1}{a-1} \times \frac{b^{j+1}-1}{b-1} \times \ldots \times \frac{c^{k+1}-1}{c-1}$. Let's try. $N = 60 = 2^2 \times 3^1 \times 5^1$, `sumDiv(60)` $= \frac{2^{2+1}-1}{2-1} \times \frac{3^{1+1}-1}{3-1} \times \frac{5^{1+1}-1}{5-1} = \frac{7 \times 8 \times 24}{1 \times 2 \times 4} = 168$.

```
ll sumDiv(ll N) {
  ll PF_idx = 0, PF = primes[PF_idx], ans = 1;   // start from ans = 1
  while (PF * PF <= N) {
    ll power = 0;
    while (N % PF == 0) { N /= PF; power++; }
    ans *= ((ll)pow((double)PF, power + 1.0) - 1) / (PF - 1);
    PF = primes[++PF_idx];
  }
  if (N != 1) ans *= ((ll)pow((double)N, 2.0) - 1) / (N - 1);  // last
  return ans;
}
```

6. `EulerPhi(N)`: Count the number of positive integers $< N$ that are relatively prime to N. Recall: Two integers a and b are said to be relatively prime (or coprime) if $gcd(a,b) = 1$, e.g. 25 and 42. A naïve algorithm to count the number of positive integers $< N$ that are relatively prime to N starts with `counter = 0`, iterates through $i \in [1..N\text{-}1]$, and increases the `counter` if $gcd(i, N) = 1$. This is slow for large N.

   A better algorithm is the Euler's Phi (Totient) function $\varphi(N) = N \times \prod_{PF} (1 - \frac{1}{PF})$, where PF is prime factor of N.

   For example $N = 36 = 2^2 \times 3^2$. $\varphi(36) = 36 \times (1 - \frac{1}{2}) \times (1 - \frac{1}{3}) = 12$. Those 12 positive integers that are relatively prime to 36 are $\{1, 5, 7, 11, 13, 17, 19, 23, 25, 29, 31, 35\}$.

```
ll EulerPhi(ll N) {
  ll PF_idx = 0, PF = primes[PF_idx], ans = N;   // start from ans = N
  while (PF * PF <= N) {
    if (N % PF == 0) ans -= ans / PF;        // only count unique factor
    while (N % PF == 0) N /= PF;
    PF = primes[++PF_idx];
  }
  if (N != 1) ans -= ans / N;                            // last factor
  return ans;
}
```

Exercise 5.5.6.1: Implement `numDiffPF(N)` and `sumPF(N)`!
Hint: Both are similar to `numPF(N)`.

5.5.7 Modified Sieve

If the number of different prime factors has to be determined for *many* (or a *range* of) integers, then there is a better solution than calling `numDiffPF(N)` as shown in Section 5.5.6 above *many times*. The better solution is the modified sieve algorithm. Instead of finding the prime factors and then calculate the required values, we start from the prime numbers and modify the values of their multiples. The short modified sieve code is shown below:

```
memset(numDiffPF, 0, sizeof numDiffPF);
for (int i = 2; i < MAX_N; i++)
  if (numDiffPF[i] == 0)                          // i is a prime number
    for (int j = i; j < MAX_N; j += i)
      numDiffPF[j]++;             // increase the values of multiples of i
```

This modified sieve algorithm should be preferred over individual calls to `numDiffPF(N)` if the range is large. However, if we just need to compute the number of different prime factors for a single but large integer N, it may be faster to just use `numDiffPF(N)`.

Exercise 5.5.7.1: The function `EulerPhi(N)` shown in Section 5.5.6 can also be re-written using modified sieve. Please write the required code!

Exercise 5.5.7.2*: Can we write the modified sieve code for the other functions listed in Section 5.5.6 above (i.e. other than `numDiffPF(N)` and `EulerPhi(N)`) without increasing the time complexity of sieve? If we can, write the required code! If we cannot, explain why!

5.5.8 Modulo Arithmetic

Some mathematical computations in programming problems can end up having very large positive (or very small negative) intermediate/final results that are beyond the range of the largest built-in integer data type (currently the 64-bit `long long` in C++ or `long` in Java). In Section 5.3, we have shown a way to compute big integers precisely. In Section 5.5.5, we have shown another way to work with big integers via its prime factors. For some other problems, we are only interested with the result *modulo* a (usually prime) number so that the intermediate/final results always fits inside built-in integer data type. In this subsection, we discuss these types of problems.

For example in UVa 10176 - Ocean Deep! Make it shallow!!, we are asked to convert a long binary number (up to 100 digits) to decimal. A quick calculation shows that the largest possible number is $2^{100} - 1$ which is beyond the range of a 64-bit integer. However, the problem only ask if the result is divisible by 131071 (which is a prime number). So what we need to do is to convert binary to decimal digit by digit, while performing modulo 131071 operation to the intermediate result. If the final result is 0, then the *actual number in binary* (which we never compute in its entirety), is divisible by 131071.

Exercise 5.5.8.1: Which statements are valid? Note: '%' is a symbol of modulo operation.
1). (a + b - c) % s = ((a % s) + (b % s) - (c % s) + s) % s
2). (a * b) % s = (a % s) * (b % s)
3). (a * b) % s = ((a % s) * (b % s)) % s
4). (a / b) % s = ((a % s) / (b % s)) % s
5). (a^b) % s = $((a^{b/2}$ % s) * $(a^{b/2}$ % s)) % s; assume that b is even.

216

5.5.9   Extended Euclid: Solving Linear Diophantine Equation

Motivating problem: Suppose a housewife buys apples and oranges with cost of 8.39 SGD. An apple is 25 cents. An orange is 18 cents. How many of each fruit does she buy?

This problem can be modeled as a linear equation with two variables: $25x + 18y = 839$. Since we know that both x and y must be integers, this linear equation is called the Linear *Diophantine* Equation. We can solve Linear Diophantine Equation with two variables even if we only have one equation! The solution is as follow:

Let a and b be integers with $d = gcd(a, b)$. The equation $ax + by = c$ has no integral solutions if $d \mid c$ is not true. But if $d \mid c$, then there are infinitely many integral solutions. The first solution (x_0, y_0) can be found using the *Extended Euclid* algorithm shown below and the rest can be derived from $x = x_0 + (b/d)n$, $y = y_0 - (a/d)n$, where n is an integer. Programming contest problems will usually have additional constraints to make the output finite (and unique).

```
// store x, y, and d as global variables
void extendedEuclid(int a, int b) {
  if (b == 0) { x = 1; y = 0; d = a; return; }            // base case
  extendedEuclid(b, a % b);                      // similar as the original gcd
  int x1 = y;
  int y1 = x - (a / b) * y;
  x = x1;
  y = y1;
}
```

Using `extendedEuclid`, we can solve the motivating problem shown earlier above: The Linear Diophantine Equation with two variables $25x + 18y = 839$.

$a = 25, b = 18$
$extendedEuclid(25, 18)$ produces $x = -5, y = 7, d = 1$; or $25 \times (-5) + 18 \times 7 = 1$.

Multiply the left and right hand side of the equation above by $839/gcd(25, 18) = 839$:
$25 \times (-4195) + 18 \times 5873 = 839$.
Thus $x = -4195 + (18/1)n$ and $y = 5873 - (25/1)n$.

Since we need to have non-negative x and y (non-negative number of apples and oranges), we have two more additional constraints:
$-4195 + 18n \geq 0$ and $5873 - 25n \geq 0$, or
$4195/18 \leq n \leq 5873/25$, or
$233.05 \leq n \leq 234.92$.

The only possible integer for n is now only 234. Thus the unique solution is $x = -4195 + 18 \times 234 = 17$ and $y = 5873 - 25 \times 234 = 23$, i.e. 17 apples (of 25 cents each) and 23 oranges (of 18 cents each) of a total of 8.39 SGD.

5.5.10   Remarks about Number Theory in Programming Contests

There are many other number theoretic problems that cannot be discussed one by one in this book. Based on our experience, number theory problems frequently appear in ICPCs especially in Asia. It is therefore a good idea for one team member to specifically study number theory listed in this book and beyond.

- Prime Numbers

  1. UVa 00406 - Prime Cuts (sieve; take the middle ones)
  2. **UVa 00543 - Goldbach's Conjecture *** (sieve; complete search; Christian Goldbach's conjecture (updated by Leonhard Euler): Every even number ≥ 4 can be expressed as the sum of two prime numbers)
  3. UVa 00686 - Goldbach's Conjecture (II) (similar to UVa 543)
  4. UVa 00897 - Annagramatic Primes (sieve; just need to check digit rotations)
  5. UVa 00914 - Jumping Champion (sieve; be careful with L and $U < 2$)
  6. **UVa 10140 - Prime Distance *** (sieve; linear scan)
  7. UVa 10168 - Summation of Four Primes (backtracking with pruning)
  8. UVa 10311 - Goldbach and Euler (case analysis, brute force, see UVa 543)
  9. **UVa 10394 - Twin Primes *** (sieve; check if p and $p + 2$ are both primes; if yes, they are twin primes; precalculate the result)
  10. UVa 10490 - Mr. Azad and his Son (Ad Hoc; precalculate the answers)
  11. UVa 10650 - Determinate Prime (sieve; find 3 uni-distance consecutive primes)
  12. UVa 10852 - Less Prime (sieve; $p = 1$, find the first prime number $\geq \frac{n}{2} + 1$)
  13. UVa 10948 - The Primary Problem (Goldbach's conjecture, see UVa 543)
  14. UVa 11752 - The Super Powers (try base: 2 to $\sqrt[4]{2^{64}}$, composite power, sort)

- GCD and/or LCM

  1. UVa 00106 - Fermat vs. Phytagoras (brute force; use GCD to get relatively prime triples)
  2. UVa 00332 - Rational Numbers from ... (use GCD to simplify fraction)
  3. UVa 00408 - Uniform Generator (cycle finding problem with easier solution: it is a good choice if `step` $<$ `mod` and `GCD(step, mod) == 1`)
  4. UVa 00412 - Pi (brute force; GCD to find elements with no common factor)
  5. **UVa 10407 - Simple Division *** (subtract the set s with s[0], find gcd)
  6. **UVa 10892 - LCM Cardinality *** (number of divisor pairs of N: (m, n) such that $gcd(m, n) = 1$)
  7. UVa 11388 - GCD LCM (understand the relationship of GCD with LCM)
  8. UVa 11417 - GCD (brute force, input is small)
  9. *UVa 11774 - Doom's Day* (find pattern involving gcd with small test cases)
  10. **UVa 11827 - Maximum GCD *** (GCD of many numbers, small input)
  11. *UVa 12068 - Harmonic Mean* (involving fraction, use LCM and GCD)

- Factorial

  1. **UVa 00324 - Factorial Frequencies *** (count digits of n! up to 366!)
  2. UVa 00568 - Just the Facts (can use Java BigInteger, slow but AC)
  3. **UVa 00623 - 500 (factorial) *** (easy with Java BigInteger)
  4. UVa 10220 - I Love Big Numbers (use Java BigInteger; precalculate)
  5. UVa 10323 - Factorial. You Must ... (overflow: n>13/-odd n; underflow: n<8/-even n; PS: actually, factorial of negative number is not defined)
  6. **UVa 10338 - Mischievous Children *** (use long long to store up to 20!)

- Finding Prime Factors

  1. **UVa 00516 - Prime Land *** (problem involving prime-power factorization)

2. **UVa 00583 - Prime Factors *** (basic prime factorization problem)

3. UVa 10392 - Factoring Large Numbers (enumerate the prime factors of input)

4. **UVa 11466 - Largest Prime Divisor *** (use efficient sieve implementation to get the largest prime factors)

- Working with Prime Factors

    1. UVa 00160 - Factors and Factorials (precalc small primes as prime factors of 100! is < 100)

    2. UVa 00993 - Product of digits (find divisors from 9 down to 1)

    3. UVa 10061 - How many zeros & how ... (in Decimal, '10' with 1 zero is due to factor 2×5)

    4. **UVa 10139 - Factovisors *** (discussed in this section)

    5. UVa 10484 - Divisibility of Factors (prime factors of factorial, D can be -ve)

    6. UVa 10527 - Persistent Numbers (similar to UVa 993)

    7. UVa 10622 - Perfect P-th Power (get GCD of all prime powers, special case if x is -ve)

    8. **UVa 10680 - LCM *** (use prime factors of `[1..N]` to get LCM(1,2,...,N))

    9. UVa 10780 - Again Prime? No time. (similar but different problem with UVa 10139)

    10. UVa 10791 - Minimum Sum LCM (analyze the prime factors of N)

    11. UVa 11347 - Multifactorials (prime-power factorization; `numDiv(N)`)

    12. *UVa 11395 - Sigma Function* (key hint: a square number multiplied by powers of two, i.e. $2^k \times i^2$ for $k \geq 0, i \geq 1$ has *odd* sum of divisors)

    13. **UVa 11889 - Benefit *** (LCM, involving prime factorization)

- Functions involving Prime Factors

    1. **UVa 00294 - Divisors *** (`numDiv(N)`)

    2. UVa 00884 - Factorial Factors (`numPF(N)`; precalculate)

    3. UVa 01246 - Find Terrorists (LA 4340, Amrita08, `numDiv(N)`)

    4. **UVa 10179 - Irreducible Basic ... *** (`EulerPhi(N)`)

    5. UVa 10299 - Relatives (`EulerPhi(N)`)

    6. UVa 10820 - Send A Table (`a[i] = a[i - 1] + 2 * EulerPhi(i)`)

    7. *UVa 10958 - How Many Solutions?* (`2 * numDiv(n * m * p * p) - 1`)

    8. UVa 11064 - Number Theory (`N - EulerPhi(N) - numDiv(N)`)

    9. UVa 11086 - Composite Prime (find numbers N with `numPF(N) == 2`)

    10. UVa 11226 - Reaching the fix-point (`sumPF(N)`; get length; DP)

    11. *UVa 11353 - A Different kind of Sorting* (`numPF(N)`; modified sorting)

    12. **UVa 11728 - Alternate Task *** (`sumDiv(N)`)

    13. *UVa 12005 - Find Solutions* (`numDiv(4N-3)`)

- Modified Sieve

    1. **UVa 10699 - Count the Factors *** (`numDiffPF(N)` for a range of N)

    2. **UVa 10738 - Riemann vs. Mertens *** (`numDiffPF(N)` for a range of N)

    3. ***UVa 10990 - Another New Function **** (modified sieve to compute a range of Euler Phi values; use DP to compute depth Phi values; then finally use Max 1D Range Sum DP to output the answer)

    4. UVa 11327 - Enumerating Rational ... (pre-calculate `EulerPhi(N)`)

    5. *UVa 12043 - Divisors* (`sumDiv(N)` and `numDiv(N)`; brute force)

- Modulo Arithmetic

  1. UVa 00128 - Software CRC $((a \times b) mod s = ((a mod s) * (b mod s)) mod s)$
  2. **UVa 00374 - Big Mod *** (solvable with Java BigInteger `modPow`; or write your own code, see Section 9.21)
  3. UVa 10127 - Ones (no factor of 2 and 5 implies that there is no trailing zero)
  4. UVa 10174 - Couple-Bachelor-Spinster ... (no Spinster number)
  5. **UVa 10176 - Ocean Deep; Make it ... *** (discussed in this section)
  6. **UVa 10212 - The Last Non-zero Digit *** (there is a modulo arithmetic solution: multiply numbers from N down to $N - M + 1$; repeatedly use $/10$ to discard the trailing zero(es), and then use %1 Billion to only memorize the last few (maximum 9) non zero digits)
  7. UVa 10489 - Boxes of Chocolates (keep working values small with modulo)
  8. *UVa 11029 - Leading and Trailing* (combination of logarithmic trick to get the first three digits and 'big mod' trick to get the last three digits)

- Extended Euclid:

  1. **UVa 10090 - Marbles *** (use solution for Linear Diophantine Equation)
  2. **UVa 10104 - Euclid Problem *** (pure problem of Extended Euclid)
  3. UVa 10633 - Rare Easy Problem (this problem can be modeled as Linear Diophantine Equation; let $C = N - M$ (the given input), $N = 10a + b$ (N is at least two digits, with b as the last digit), and $M = a$; this problem is now about finding the solution of the Linear Diophantine Equation: $9a + b = C$)
  4. **UVa 10673 - Play with Floor and Ceil *** (uses Extended Euclid)

- Other Number Theory Problems

  1. UVa 00547 - DDF (a problem about 'eventually constant' sequence)
  2. UVa 00756 - Biorhythms (Chinese Remainder Theorem)
  3. **UVa 10110 - Light, more light *** (check if n is a square number)
  4. UVa 10922 - 2 the 9s (test divisibility by 9)
  5. UVa 10929 - You can say 11 (test divisibility by 11)
  6. UVa 11042 - Complex, difficult and ... (case analysis; only 4 possible outputs)
  7. **UVa 11344 - The Huge One *** (read M as string, use divisibility theory of `[1..12]`)
  8. **UVa 11371 - Number Theory for ... *** (the solving strategy is given)

Profile of Algorithm Inventors

John Pollard (born 1941) is a British mathematician who has invented algorithms for the factorization of large numbers (the Pollard's rho algorithm) and for the calculation of discrete logarithms (not discussed in this book).

Richard Peirce Brent (born 1946) is an Australian mathematician and computer scientist. His research interests include number theory (in particular factorization), random number generators, computer architecture, and analysis of algorithms. He has invented or co-invented various mathematics algorithms. In this book, we discuss Brent's cycle-finding algorithm (see **Exercise 5.7.1***) and Brent's improvement of the Pollard's rho algorithm (see **Exercise 5.5.4.2*** and Section 9.26).

5.6 Probability Theory

Probability Theory is a branch of mathematics dealing with the analysis of random phenomena. Although an event like an individual (fair) coin toss is random, the sequence of random events will exhibit certain statistical patterns if the event is repeated many times. This can be studied and predicted. The probability of a head appearing is 1/2 (similarly with a tail). Therefore, if we flip a (fair) coin n times, we *expect* that we see heads $n/2$ times.

In programming contests, problems involving probability are either solvable with:

- Closed-form formula. For these problems, one has to derive the required (usually $O(1)$) formula. For example, let's discuss how to derive the solution for UVa 10491 - Cows and Cars, which is a generalized version of a TV show: 'The Monty Hall problem'[15].

  You are given NCOWS number of doors with cows, NCARS number of doors with cars, and NSHOW number of doors (with cows) that are opened for you by the presenter. Now, you need to count the probability of winning a car assuming that you will always switch to another unopened door.

  The first step is to realize that there are two ways to get a car. Either you pick a cow first and then switch to a car, or you pick a car first, and then switch to another car. The probability of each case can be computed as shown below.

  In the first case, the chance of picking a cow first is (NCOWS / (NCOWS+NCARS)). Then, the chance of switching to a car is (NCARS / (NCARS+NCOWS-NSHOW-1)). Multiply these two values together to get the probability of the first case. The -1 is to account for the door that you have already chosen, as you cannot switch to it.

  The probability of the second case can be computed in a similar manner. The chance of picking a car first is (NCARS / (NCARS+NCOWS)). Then, the chance of switching to a car is ((NCARS-1) / (NCARS+NCOWS-NSHOW-1)). Both -1 accounts for the car that you have already chosen.

  Sum the probability values of these two cases together to get the final answer.

- Exploration of the search (sample) space to count number of events (usually harder to count; may deal with combinatorics—see Section 5.4, Complete Search—see Section 3.2, or Dynamic Programming—see Section 3.5) over the countable sample space (usually much simpler to count). Examples:

  - 'UVa 12024 - Hats' is a problem of n people who store their n hats in a cloakroom for an event. When the event is over, these n people take their hats back. Some take a wrong hat. Compute how likely is that *everyone* take a wrong hat?

    This problem can be solved via brute-force and pre-calculation by trying all $n!$ permutations and see how many times the required events appear over $n!$ because $n \leq 12$ in this problem. However, a more math-savvy contestant can use this Derangement (DP) formula instead: $A_n = (n-1) \times (A_{n-1} + A_{n-2})$.

  - 'UVa 10759 - Dice Throwing' has a short description: n common cubic dice are thrown. What is the probability that the sum of all thrown dices is at least x? (constraints: $1 \leq n \leq 24$, $0 \leq x < 150$).

---

[15]This is an interesting probability puzzle. Readers who have not heard this problem before is encouraged to do some Internet search and read the history of this problem. In the original problem, NCOWS = 2, NCARS = 1, and NSHOW = 1. The probability of staying with your original choice is $\frac{1}{3}$ and the probability of switching to another unopened door is $\frac{2}{3}$ and therefore it is always beneficial to switch.

The sample space (the denominator of the probability value) is very simple to compute. It is 6^n.

The number of events is slightly harder to compute. We need a (simple) DP because there are lots of overlapping subproblems. The state is $(dice\_left, score)$ where $dice\_left$ keeps track of the remaining dice that we can still throw (starting from n) and $score$ counts the accumulated score so far (starting from 0). DP can be used as there are only $24 \times (24 \times 6) = 3456$ distinct states for this problem.

When $dice\_left = 0$, we return 1 (event) if $score \geq x$, or return 0 otherwise; When $dice\_left > 0$, we try throwing one more dice. The outcome v for this dice can be one of six values and we move to state $(dice\_left - 1, score + v)$. We sum all the events.

One final requirement is that we have to use gcd (see Section 5.5.2) to simplify the probability fraction. In some other problems, we may be asked to output the probability value correct to a certain digit after decimal point.

Programming Exercises about Probability Theory:

1. *UVa 00542 - France '98* (divide and conquer)
2. UVa 10056 - What is the Probability? (get the closed form formula)
3. *UVa 10218 - Let's Dance* (probability and a bit of binomial coefficients)
4. UVa 10238 - Throw the Dice (similar to UVa 10759; use Java BigInteger)
5. UVa 10328 - Coin Toss (DP, 1-D state, Java BigInteger)
6. **UVa 10491 - Cows and Cars *** (discussed in this section)
7. **UVa 10759 - Dice Throwing *** (discussed in this section)
8. *UVa 10777 - God, Save me* (expected value)
9. *UVa 11021 - Tribbles* (probability)
10. ***UVa 11176 - Winning Streak **** (DP, s: (n_left, max_streak) where n_left is the number of remaining games and max_streak stores the longest consecutive wins; t: lose this game, or win the next W = [1..n_left] games and lose the (W+1)-th game; special case if W = n_left)
11. UVa 11181 - Probability (bar) Given (iterative brute force, try all possibilities)
12. *UVa 11346 - Probability* (a bit of geometry)
13. UVa 11500 - Vampires (Gambler's Ruin Problem)
14. UVa 11628 - Another lottery (p[i] = ticket[i] / total; use gcd to simplify fraction)
15. UVa 12024 - Hats (discussed in this section)
16. *UVa 12114 - Bachelor Arithmetic* (simple probability)
17. *UVa 12457 - Tennis contest* (simple expected value problem; use DP)
18. *UVa 12461 - Airplane* (brute force small n to see that the answer is very easy)

5.7 Cycle-Finding

Given a function $f : S \to S$ (that maps a natural number from a *finite set S* to another natural number in the same finite set S) and an initial value $x_0 \in N$, the sequence of **iterated function values**: $\{x_0, x_1 = f(x_0), x_2 = f(x_1), \ldots, x_i = f(x_{i-1}), \ldots\}$ must eventually use the same value twice, i.e. $\exists i \neq j$ such that $x_i = x_j$. Once this happens, the sequence must then repeat the cycle of values from x_i to x_{j-1}. Let μ (the start of cycle) be the smallest index i and λ (the cycle length) be the smallest positive integer such that $x_\mu = x_{\mu+\lambda}$. The **cycle-finding** problem is defined as the problem of finding μ and λ given $f(x)$ and x_0.

For example in UVa 350 - Pseudo-Random Numbers, we are given a pseudo-random number generator $f(x) = (Z \times x + I)\%M$ with $x_0 = L$ and we want to find out the sequence length before any number is repeated (i.e. the λ). A good pseudo-random number generator should have a large λ. Otherwise, the numbers generated will not look 'random'.

Let's try this process with the sample test case $Z = 7, I = 5, M = 12, L = 4$, so we have $f(x) = (7 \times x + 5)\%12$ and $x_0 = 4$. The sequence of iterated function values is $\{4, 9, 8, 1, 0, 5, \underline{4, 9, 8, 1, 0, 5}, \ldots\}$. We have $\mu = 0$ and $\lambda = 6$ as $x_0 = x_{\mu+\lambda} = x_{0+6} = x_6 = 4$. The sequence of iterated function values cycles from index 6 onwards.

On another test case $Z = 3, I = 1, M = 4, L = 7$, we have $f(x) = (3 \times x + 1)\%4$ and $x_0 = 7$. The sequence of iterated function values is $\{7, 2, 3, \underline{2, 3}, \ldots\}$. This time, we have $\mu = 1$ and $\lambda = 2$.

5.7.1 Solution(s) using Efficient Data Structure

A simple algorithm that will work for *many cases* of this cycle-finding problem uses an efficient data structure to store pair of information that a number x_i has been encountered at iteration i in the sequence of iterated function values. Then for x_j that is encountered later $(j > i)$, we test if x_j is already stored in the data structure. If it is, it implies that $x_j = x_i$, $\mu = i$, $\lambda = j - i$. This algorithm runs in $O((\mu + \lambda) \times DS\_cost)$ where DS_cost is the cost per one data structure operation (insert/search). This algorithm requires at least $O(\mu + \lambda)$ space to store past values.

For many cycle-finding problems with rather large S (and likely large $\mu + \lambda$), we can use $O(\mu + \lambda)$ space C++ STL `map`/Java `TreeMap` to store/check the iteration indices of past values in $O(\log(\mu + \lambda))$ time. But if we just need to stop the algorithm upon encountering the *first* repeated number, we can use C++ STL `set`/Java `TreeSet` instead.

For other cycle-finding problems with relatively small S (and likely small $\mu + \lambda$), we may use the $O(|S|)$ space Direct Addressing Table to store/check the iteration indices of past values in $O(1)$ time. Here, we trade-off memory space for runtime speed.

5.7.2 Floyd's Cycle-Finding Algorithm

There is a better algorithm called Floyd's cycle-finding algorithm that runs in $O(\mu + \lambda)$ time complexity and *only* uses $O(1)$ memory space—much smaller than the simple versions shown above. This algorithm is also called 'the tortoise and hare (rabbit)' algorithm. It has three components that we describe below using the UVa 350 problem as shown above with $Z = 3, I = 1, M = 4, L = 7$.

Efficient Way to Detect a Cycle: Finding $k\lambda$

Observe that for any $i \geq \mu$, $x_i = x_{i+k\lambda}$, where $k > 0$, e.g. in Table 5.2, $x_1 = x_{1+1\times2} = x_3 = x_{1+2\times2} = x_5 = 2$, and so on. If we set $k\lambda = i$, we get $x_i = x_{i+i} = x_{2i}$. Floyd's cycle finding algorithm exploits this trick.

step	x_0	x_1	x_2	x_3	x_4	x_5	x_6
	7	2	3	2	3	2	3
Init	TH						
1		T	H				
2			T		H		

Table 5.2: Part 1: Finding $k\lambda$, $f(x) = (3 \times x + 1)\%4$, $x_0 = 7$

The Floyd's cycle-finding algorithm maintains two pointers called 'tortoise' (the slower one) at x_i and 'hare' (the faster one that keeps jumping around) at x_{2i}. Initially, both are at x_0. At each step of the algorithm, tortoise is moved *one step* to the right and the hare is moved *two steps* to the right[16] in the sequence. Then, the algorithm compares the sequence values at these two pointers. The smallest value of $i > 0$ for which both tortoise and hare point to equal values is the value of $k\lambda$ (multiple of λ). We will determine the actual λ from $k\lambda$ using the next two steps. In Table 5.2, when $i = 2$, we have $x_2 = x_4 = x_{2+\mathbf{2}} = x_{2+k\lambda} = 3$. So, $k\lambda = 2$. In this example, we will see below that k is eventually 1, so $\lambda = 2$ too.

Finding μ

Next, we reset hare back to x_0 and keep tortoise at its current position. Now, we advance *both* pointers to the right one step at a time, thus maintaining the $k\lambda$ gap between the two pointers. When tortoise and hare points to the same value, we have just found the *first* repetition of length $k\lambda$. Since $k\lambda$ is a multiple of λ, it must be true that $x_\mu = x_{\mu+k\lambda}$. The first time we encounter the first repetition of length $k\lambda$ is the value of the μ. In Table 5.3, we find that $\mu = 1$.

step	x_0	x_1	x_2	x_3	x_4	x_5	x_6
	7	2	3	2	3	2	3
1	H		T				
2		H		T			

Table 5.3: Part 2: Finding μ

Finding λ

Once we get μ, we let tortoise stays in its current position and set hare next to it. Now, we move hare iteratively to the right one by one. Hare will point to a value that is the same as tortoise for the *first* time after λ steps. In Table 5.4, after hare moves once, $x_3 = x_{3+2} = x_5 = 2$. So, $\lambda = 2$.

step	x_0	x_1	x_2	x_3	x_4	x_5	x_6
	7	2	3	2	3	2	3
1				T	H		
2				T		H	

Table 5.4: Part 3: Finding λ

Therefore, we report $\mu = 1$ and $\lambda = 2$ for $f(x) = (3 \times x + 1)\%4$ and $x_0 = 7$.
In overall, this algorithm runs in $O(\mu + \lambda)$.

---

[16]To move right one step from x_i, we use $x_i = f(x_i)$. To move right two steps from x_i, we use $x_i = f(f(x_i))$.

The Implementation

The working C/C++ implementation of this algorithm (with comments) is shown below:

```
ii floydCycleFinding(int x0) {  // function int f(int x) is defined earlier
  // 1st part: finding k*mu, hare's speed is 2x tortoise's
  int tortoise = f(x0), hare = f(f(x0));    // f(x0) is the node next to x0
  while (tortoise != hare) { tortoise = f(tortoise); hare = f(f(hare)); }
  // 2nd part: finding mu, hare and tortoise move at the same speed
  int mu = 0; hare = x0;
  while (tortoise != hare) { tortoise = f(tortoise); hare = f(hare); mu++;}
  // 3rd part: finding lambda, hare moves, tortoise stays
  int lambda = 1; hare = f(tortoise);
  while (tortoise != hare) { hare = f(hare); lambda++; }
  return ii(mu, lambda);
}
```

Source code: `ch5_07_UVa350.cpp/java`

Exercise 5.7.1*: Richard P. Brent invented an improved version of Floyd's cycle-finding algorithm shown above. Study and implement Brent's algorithm [3]!

Programming Exercises related to Cycle-Finding:

1. UVa 00202 - Repeating Decimals (do expansion digit by digit until it cycles)
2. UVa 00275 - Expanding Fractions (same as UVa 202 except the output format)
3. **UVa 00350 - Pseudo-Random Numbers *** (discussed in this section)
4. UVa 00944 - Happy Numbers (similar to UVa 10591)
5. UVa 10162 - Last Digit (cycle after 100 steps, use Java BigInteger to read the input, precalculate)
6. UVa 10515 - Power et al (concentrate on the last digit)
7. UVa 10591 - Happy Number (this sequence is 'eventually periodic')
8. UVa 11036 - Eventually periodic ... (cycle-finding, evaluate Reverse Polish `f` with a `stack`—also see Section 9.27)
9. **UVa 11053 - Flavius Josephus ... *** (cycle-finding, the answer is $N - \lambda$)
10. UVa 11549 - Calculator Conundrum (repeat squaring with limited digits until it cycles; that is, the Floyd's cycle-finding algorithm is only used to detect the cycle, we do not use the value of μ or λ; instead, we keep track the largest iterated function value found before any cycle is encountered)
11. **UVa 11634 - Generate random ... *** (use DAT of size 10K, extract digits; the programming trick to square 4 digits 'a' and get the resulting middle 4 digits is `a = (a * a / 100) % 10000`)
12. *UVa 12464 - Professor Lazy, Ph.D.* (although n can be very huge, the pattern is actually cyclic; find the length of the cycle l and modulo n with l)

5.8 Game Theory

Game Theory is a mathematical model of strategic situations (not necessarily *games* as in the common meaning of 'games') in which a player's success in making choices depends on the choices of *others*. Many programming problems involving game theory are classified as **Zero-Sum Game**—a mathematical way of saying that if one player wins, then the other player loses. For example, a game of Tic-Tac-Toe (e.g. UVa 10111), Chess, various number/integer games (e.g. UVa 847, 10368, 10578, 10891, 11489), and others (UVa 10165, 10404, 11311) are games with two players playing alternately (usually perfectly) and there can only be one winner.

The common question asked in programming contest problems related to game theory is whether the starting player of a two player competitive game has a winning move assuming that both players are doing **Perfect Play**. That is, each player always choose the most optimal choice available to him.

5.8.1 Decision Tree

One solution is to write a recursive code to explore the **Decision Tree** of the game (a.k.a. the Game Tree). If there is no overlapping subproblem, pure recursive backtracking is suitable. Otherwise, Dynamic Programming is needed. Each vertex describes the current player and the current state of the game. Each vertex is connected to all other vertices legally reachable from that vertex according to the game rules. The root vertex describes the starting player and the initial game state. If the game state at a leaf vertex is a winning state, it is a win for the current player (and a lose for the other player). At an internal vertex, the current player chooses a vertex that guarantees a win with the largest margin (or if a win is not possible, choose a vertex with the least loss). This is called the **Minimax** strategy.

For example, in UVa 10368 - Euclid's Game, there are two players: Stan (player 0) and Ollie (player 1). The state of the game is a triple of integers (id, a, b). The current player id can subtracts any positive multiple of the lesser of the two numbers, integer b, from the greater of the two numbers, integer a, provided that the resulting number must be nonnegative. We always maintain that $a \geq b$. Stan and Ollie plays alternately, until one player is able to subtract a multiple of the lesser number from the greater to reach 0, and thereby wins. The first player is Stan. The decision tree for a game with initial state $id = 0$, $a = 34$, and $b = 12$ is shown below in Figure 5.2 .
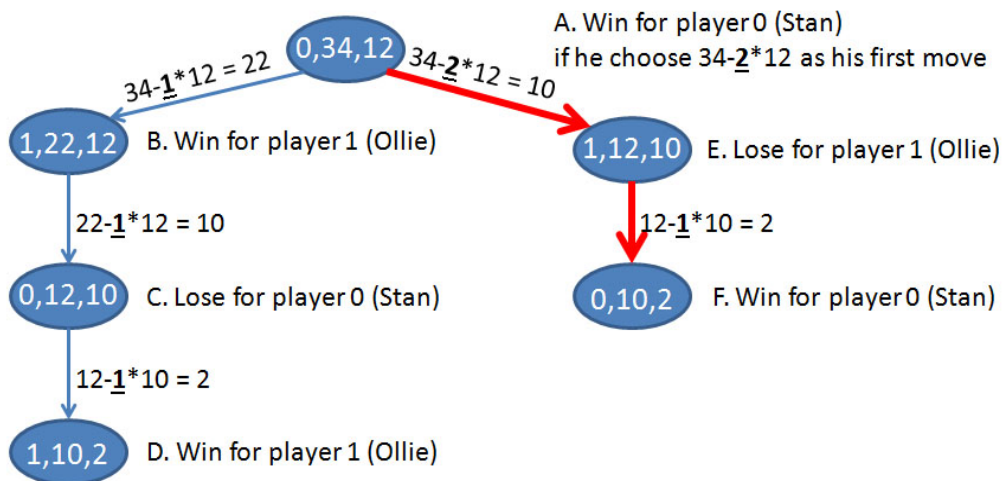
Figure 5.2: Decision Tree for an instance of 'Euclid's Game'

Let's trace what happens in Figure 5.2. At the root (initial state), we have triple $(0, 34, 12)$. At this point, player 0 (Stan) has two choices: Either to subtract $a - b = 34 - 12 = 22$ and move to vertex $(1, 22, 12)$ (the left branch) or to subtract $a - 2 \times b = 24 - 2 \times 12 = 10$ and move to vertex $(1, 12, 10)$ (the right branch). We try both choices recursively.

Let's start with the left branch. At vertex $(1, 22, 12)$—(Figure 5.2.B), the current player 1 (Ollie) has no choice but to subtract $a - b = 22 - 12 = 10$. We are now at vertex $(0, 12, 10)$—(Figure 5.2.C). Again, Stan only has one choice which is to subtract $a - b = 12 - 10 = 2$. We are now at leaf vertex $(1, 10, 2)$—(Figure 5.2.D). Ollie has several choices but Ollie can definitely win as $a - 5 \times b = 10 - 5 \times 2 = 0$ and it implies that vertex $(0, 12, 10)$ is a losing state for Stan and vertex $(1, 22, 12)$ is a winning state for Ollie.

Now we explore the right branch. At vertex $(1, 12, 10)$—(Figure 5.2.E), the current player 1 (Ollie) has no choice but to subtract $a - b = 12 - 10 = 2$. We are now at leaf vertex $(0, 10, 2)$—(Figure 5.2.F). Stan has several choices but Stan can definitely win as $a - 5 \times b = 10 - 5 \times 2 = 0$ and it implies that vertex $(1, 12, 10)$ is a losing state for Ollie.

Therefore, for player 0 (Stan) to win this game, Stan should choose $a - 2 \times b = 34 - 2 \times 12$ first, as this is a winning move for Stan—(Figure 5.2.A).

Implementation wise, the first integer id in the triple can be dropped as we know that depth 0 (root), 2, 4, ... are always Stan's turns and depth 1, 3, 5, ... are always Ollie's turns. This integer id is used in Figure 5.2 to simplify the explanation.

5.8.2 Mathematical Insights to Speed-up the Solution

Not all game theory problems can be solved by exploring the *entire* decision tree of the game, especially if the size of the tree is large. If the problem involves numbers, we may need to come up with some mathematical insights to speed up the computation.

For example, in UVa 847 - A multiplication game, there are two players: Stan (player 0) and Ollie (player 1) again. The state of the game[17] is an integer p. The current player can multiply p with any number between 2 to 9. Stan and Ollie also plays alternately, until one player is able to multiply p with a number between 2 to 9 such that $p \geq n$ (n is the target number), thereby wins. The first player is Stan with $p = 1$.

Figure 5.3 shows an instance of this multiplication game with $n = 17$. Initially, player 0 has up to 8 choices (to multiply $p = 1$ by $[2..9]$). However, all of these 8 states are winning states of player 1 as player 1 can always multiply the current p by $[2..9]$ to make $p \geq 17$—(Figure 5.3.B). Therefore player 0 will surely lose—(Figure 5.3.A).
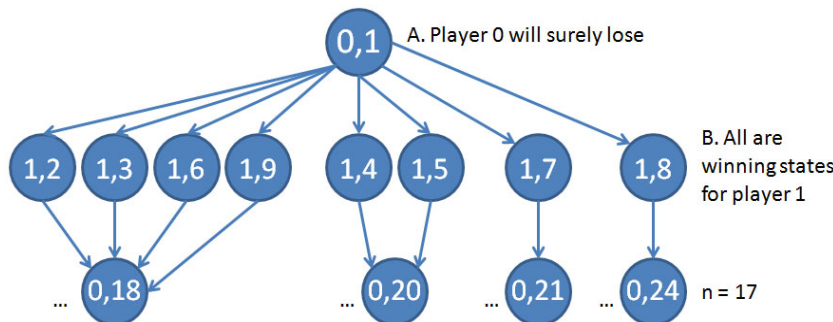
Figure 5.3: Partial Decision Tree for an instance of 'A multiplication game'

As $1 < n < 4294967295$, the resulting decision tree on the largest test case can be extremely huge. This is because each vertex in this decision tree has a *huge* branching factor of 8 (as

---

[17]This time we omit the player id. However, this parameter id is still shown in Figure 5.3 for clarity.

there are 8 possible numbers to choose from between 2 to 9). It is not feasible to actually explore the decision tree.

It turns out that the optimal strategy for Stan to win is to *always* multiply p with 9 (the largest possible) while Ollie will *always* multiply p with 2 (the smallest possible). Such optimization insights can be obtained by observing the pattern found in the output of smaller instances of this problem. Note that a maths-savvy contestants may want to prove this observation first before coding the solution.

5.8.3   Nim Game

There is a special game that is worth mentioning as it may appear in programming contests: The **Nim** game[18]. In Nim game, two players take turns to remove objects from distinct heaps. On each turn, a player must remove *at least one object* and may remove *any number of objects* provided they all come from the same heap. The initial state of the game is the number of objects n_i at each of the k heaps: $\{n_1, n_2, \ldots, n_k\}$. There is a nice solution for this game. For the first (starting) player to win, the value of $n_1 \text{ ^ } n_2 \text{ ^ } \ldots \text{ ^ } n_k$ must be *non zero* where $^\wedge$ is the bit operator xor (exclusive or)—proof omitted.

Programming Exercises related to Game Theory:

1. UVa 00847 - A multiplication game (simulate the perfect play, discussed above)

2. **UVa 10111 - Find the Winning ... *** (Tic-Tac-Toe, minimax, backtracking)

3. UVa 10165 - Stone Game (Nim game, application of Sprague-Grundy theorem)

4. UVa 10368 - Euclid's Game (minimax, backtracking, discussed in this section)

5. UVa 10404 - Bachet's Game (2 players game, Dynamic Programming)

6. UVa 10578 - The Game of 31 (backtracking; try all; see who wins the game)

7. **UVa 11311 - Exclusively Edible *** (game theory, reducible to Nim game; we can view the game that Handel and Gretel are playing as Nim game, where there are 4 heaps - cakes left/below/right/above the topping; take the Nim sum of these 4 values and if they are equal to 0, Handel loses)

8. **UVa 11489 - Integer Game *** (game theory, reducible to simple math)

9. *UVa 12293 - Box Game* (analyze the game tree of smaller instances to get the mathematical insight to solve this problem)

10. *UVa 12469 - Stones* (game playing, Dynamic Programming, pruning)

---

[18]The general form of two player games is inside the IOI syllabus [20], but Nim game is not.

5.9 Solution to Non-Starred Exercises

Exercise 5.2.1: The $<$cmath$>$ library in C/C++ has two functions: `log` (base e) and `log10` (base 10); `Java.lang.Math` only has `log` (base e). To get `log`$_b$`(a)` (base b), we use the fact that `log`$_b$`(a)` = `log(a) / log(b)`.

Exercise 5.2.2: `(int)floor(1 + log10((double)a))` returns the number of digits in decimal number `a`. To count the number of digits in other base `b`, we can use similar formula: `(int)floor(1 + log10((double)a) / log10((double)b))`.

Exercise 5.2.3: $\sqrt[n]{a}$ can be rewritten as $a^{1/n}$. We can then use built in formula like `pow((double)a, 1.0 / (double)n)` or `exp(log((double)a) * 1.0 / (double)n)`.

Exercise 5.3.1.1: Possible, keep the intermediate computations **modulo** 10^6. Keep chipping away the trailing zeroes (either none or a few zeroes are added after a multiplication from $n!$ to $(n+1)!$).

Exercise 5.3.1.2: Possible. $9317 = 7 \times 11^3$. We also list 25! as its prime factors. Then, we check if there are one factor 7 (yes) and three factors 11 (unfortunately no). So 25! is not divisible by 9317. Alternative approach: Use modulo arithmetic (see Section 5.5.8).

Exercise 5.3.2.1: For base number conversion of 32-bit integers, use `parseInt(String s, int radix)` and `toString(int i, int radix)` in the faster Java **Integer** class. You can also use `BufferedReader` and `BufferedWriter` for I/O (see Section 3.2.3).

Exercise 5.4.1.1: Binet's closed-form formula for Fibonacci: $fib(n) = (\phi^n - (-\phi)^{-n})/\sqrt{5}$ should be correct for larger n. But since double precision data type is limited, we have discrepancies for larger n. This closed form formula is correct up to $fib(75)$ if implemented using typical double data type in a computer program. This is unfortunately too small to be useful in typical programming contest problems involving Fibonacci numbers.

Exercise 5.4.2.1: $C(n,2) = \frac{n!}{(n-2)! \times 2!} = \frac{n \times (n-1) \times (n-2)!}{(n-2)! \times 2} = \frac{n \times (n-1)}{2} = 0.5n^2 - 0.5n = O(n^2)$.

Exercise 5.4.4.1: Fundamental counting principle: If there are m ways to do one thing and n ways to do another thing, then there are $m \times n$ ways to do both. The answer for this exercise is therefore: $6 \times 6 \times 2 \times 2 = 6^2 \times 2^2 = 36 \times 4 = 144$ different possible outcomes.

Exercise 5.4.4.2: See above. The answer is: $9 \times 9 \times 8 = 648$. Initially there are 9 choices (1-9), then there are still 9 choices (1-9 minus 1, plus 0), then finally there are only 8 choices.

Exercise 5.4.4.3: A permutation is an arrangement of objects without repetition and the order is important. The formula is $_nP_r = \frac{n!}{(n-r)!}$. The answer for this exercise is therefore: $\frac{6!}{(6-3)!} = 6 \times 5 \times 4 = 120$ 3-letters words.

Exercise 5.4.4.4: The formula to count different permutations is: $\frac{n!}{(n_1)! \times (n_2)! \times ... \times (n_k)!}$ where n_i is the frequency of each unique letter i and $n_1 + n_2 + ... + n_k = n$. The answer for this exercise is therefore: $\frac{5!}{3! \times 1! \times 1!} = \frac{120}{6} = 20$ because there are 3 'B's, 1 'O', and 1 'Y'.

Exercise 5.4.4.5: The answers for few small $n = 3$, 4, 5, 6, 7, 8, 9, and 10 are 0, 1, 3, 7, 13, 22, 34, and 50, respectively. You can generate these numbers using brute force solution first. Then find the pattern and use it.

Exercise 5.5.2.1: Multiplying $a \times b$ first before dividing the result by $gcd(a, b)$ has a higher chance of overflow in programming contest than $a \times (b/gcd(a, b))$. In the example given, we have $a = 1000000000$ and $b = 8$. The LCM is 1000000000—which should fit in 32-bit signed integers—can only be properly computed with $a \times (b/gcd(a, b))$.

Exercise 5.5.4.1: Since the largest prime in `vi` 'primes' is 9999991, this code can therefore

handles $N \leq 9999991^2 = 99999820000081 \approx 9 \times 10^{13}$. If the smallest prime factor of N is greater than 9999991, for example, $N = 1010189899^2 = 1020483632041630201 \approx 1 \times 10^{18}$ (this still within the capacity of 64-bit signed integer), this code will crash or produce wrong result. If we decide to drop the usage of vi 'primes' and uses $PF = 3, 5, 7, \ldots$ (with special check for $PF = 2$), then we have a slower code and the newer limit for N is now N with smallest prime factor up to $2^{63} - 1$. However, if such input is given, we need to use the algorithms mentioned in **Exercise 5.5.4.2*** and in Section 9.26.

Exercise 5.5.4.2: See Section 9.26.

Exercise 5.5.5.1: GCD(A, B) can be obtained by taking the lower power of the common prime factors of A and B. LCM(A, B) can be obtained by taking the greater power of all the prime factors of A and B. So, GCD($2^6 \times 3^3 \times 97^1$, $2^5 \times 5^2 \times 11^2$) = 2^5 = 32 and LCM($2^6 \times 3^3 \times 97^1$, $2^5 \times 5^2 \times 11^2$) = $2^6 \times 3^3 \times 5^2 \times 11^2 \times 97^1$ = 507038400.

Exercise 5.5.6.1:

```
ll numDiffPF(ll N) {
  ll PF_idx = 0, PF = primes[PF_idx], ans = 0;
  while (PF * PF <= N) {
    if (N % PF == 0) ans++;                     // count this pf only once
    while (N % PF == 0) N /= PF;
    PF = primes[++PF_idx];
  }
  if (N != 1) ans++;
  return ans;
}
```

```
ll sumPF(ll N) {
  ll PF_idx = 0, PF = primes[PF_idx], ans = 0;
  while (PF * PF <= N) {
    while (N % PF == 0) { N /= PF; ans += PF; }
    PF = primes[++PF_idx];
  }
  if (N != 1) ans += N;
  return ans;
}
```

Exercise 5.5.7.1: The modified sieve code to compute the Euler Totient function up to 10^6 is shown below:

```
for (int i = 1; i <= 1000000; i++) EulerPhi[i] = i;
for (int i = 2; i <= 1000000; i++)
  if (EulerPhi[i] == i)                          // i is a prime number
    for (int j = i; j <= 1000000; j += i)
      EulerPhi[j] = (EulerPhi[j] / i) * (i - 1);
```

Exercise 5.5.8.1: Statement 2 and 4 are not valid. The other 3 are valid.

5.10 Chapter Notes

This chapter has grown significantly since the first edition of this book. However, even until the third edition, we become more aware that there are still many more mathematics problems and algorithms that have not been discussed in this chapter, e.g.

- There are many more but rare **combinatorics** problems and formulas that are not yet discussed: **Burnside's lemma**, **Stirling Numbers**, etc.

- There are other theorems, hypothesis, and conjectures that cannot be discussed one by one, e.g. **Carmichael's function**, **Riemann's hypothesis**, **Fermat's Little Test**, **Chinese Remainder Theorem**, **Sprague-Grundy Theorem**, etc.

- We only briefly mention Brent's cycle-finding algorithm (that is slightly faster than Floyd's version) in **Exercise 5.7.1***.

- (Computational) Geometry is also part of Mathematics, but since we have a special chapter for that, we reserve the discussions about geometry problems in Chapter 7.

- Later in Chapter 9, we briefly discuss a few more mathematics-related algorithm, e.g. Gaussian Elimination for solving system of linear equations (Section 9.9), Matrix Power and its usages (Section 9.21), Pollard's rho algorithm (Section 9.26), Postfix Calculator and (Infix to Postfix) Conversion and (Section 9.27), Roman Numerals (Section 9.28).

There are really *many* topics about mathematics. This is not surprising since various mathematics problems have been investigated by people since hundreds years ago. Some of them are discussed in this chapter, many others are not, and yet only 1 or 2 will actually appear in a problem set. To do well in ICPC, it is a good idea to have at least *one strong mathematician* in your ICPC team in order to have those 1 or 2 mathematics problems solved. Mathematical prowess is also important for IOI contestants. Although the amount of problem-specific topics to be mastered is smaller, many IOI tasks require some form of 'mathematical insights'.

We end this chapter by listing some pointers that may be of interest to some readers: Read number theory books, e.g. [56], investigate mathematical topics in `mathworld.wolfram.com` or Wikipedia, and attempt many more programming exercises related to mathematics problems like the ones in `http://projecteuler.net` [17] and `https://brilliant.org` [4].

Statistics	First Edition	Second Edition	Third Edition
Number of Pages	17	29 (+71%)	41 (+41%)
Written Exercises	-	19	20+10*=30 (+58%)
Programming Exercises	175	296 (+69%)	369 (+25%)

The breakdown of the number of programming exercises from each section is shown below:

Section	Title	Appearance	% in Chapter	% in Book
5.2	**Ad Hoc Mathematics ...**	144	39%	9%
5.3	Java BigInteger Class	45	12%	3%
5.4	Combinatorics	54	15%	3%
5.5	**Number Theory**	86	23%	5%
5.6	Probability Theory	18	5%	1%
5.7	Cycle-Finding	13	3%	1%
5.8	Game Theory	10	3%	1%

L-R: Steven, Ranald, Hubert, Wei Liang, Bernard, Zi Chun

Chapter 6

String Processing

The Human Genome has approximately 3.2 Giga base pairs
— **Human Genome Project**

6.1 Overview and Motivation

In this chapter, we present one more topic that is tested in ICPC—although not as frequent[1] as graph and mathematics problems—namely: String processing. String processing is common in the research field of *bioinformatics*. As the strings (e.g. DNA strings) that researchers deal with are usually (very) long, efficient string-specific data structures and algorithms are necessary. Some of these problems are presented as contest problems in ICPCs. By mastering the content of this chapter, ICPC contestants will have a better chance at tackling those string processing problems.

String processing tasks also appear in IOI, but usually they do not require advanced string data structures or algorithms due to syllabus [20] restriction. Additionally, the input and output format of IOI tasks are usually simple[2]. This eliminates the need to code tedious input parsing or output formatting commonly found in ICPC problems. IOI tasks that require string processing are usually still solvable using the problem solving paradigms mentioned in Chapter 3. It is sufficient for IOI contestants to skim through all sections in this chapter except Section 6.5 about string processing with DP. However, we believe that it may be advantageous for IOI contestants to learn some of the more advanced materials outside of their syllabus ahead of time.

This chapter is structured as follows: It starts with an overview of basic string processing skills and a *long* list of Ad Hoc string problems solvable with that basic string processing skills. Although the Ad Hoc string problems constitute the majority of the problems listed in this chapter, we have to make a remark that recent contest problems in ACM ICPC (and also IOI) usually do not ask for basic string processing solutions *except* for the 'giveaway' problem that most teams (contestants) should be able to solve. The more important sections are the string matching problems (Section 6.4), string processing problems solvable with Dynamic Programming (DP) (Section 6.5), and finally an extensive discussion on string processing problems where we have to deal with reasonably **long** strings (Section 6.6). The last section involves a discussion on an efficient data structure for strings like Suffix **Trie**, Suffix **Tree**, and Suffix **Array**.

---

[1]One potential reason: String input is harder to parse correctly and string output is harder to format correctly, making such string-based I/O less preferred over the more precise integer-based I/O.

[2]IOI 2010-2012 require contestants to implement functions instead of coding I/O routines.

6.2 Basic String Processing Skills

We begin this chapter by listing several *basic* string processing skills that every competitive programmer must have. In this section, we give a series of mini tasks that you should solve one after another without skipping. You can use any of the three programming languages: C, C++, and/or Java. Try your best to come up with the shortest, most efficient implementation that you can think of. Then, compare your implementations with ours (see the answers at the back of this chapter). If you are not surprised with any of our implementations (or can even give simpler implementations), then you are already in a good shape for tackling various string processing problems. Go ahead and read the next sections. Otherwise, please spend some time studying our implementations.

1. Given a text file that contains only alphabet characters `[A-Za-z]`, digits `[0-9]`, space, and period (`‘.’`), write a program to read this text file line by line until we encounter a line that *starts with* seven periods (`‘‘.......’’`). Concatenate (combine) each line into one long string `T`. When two lines are combined, give one space between them so that the last word of the previous line is separated from the first word of the current line. There can be up to 30 characters per line and no more than 10 lines for this input block. There is no trailing space at the end of each line and each line ends with a newline character. Note: The sample input text file `‘ch6.txt’` is shown inside a box after question 1.(d) and before task 2.

   (a) Do you know how to store a string in your favorite programming language?
   (b) How to read a given text input line by line?
   (c) How to concatenate (combine) two strings into a larger one?
   (d) How to check if a line starts with a string `‘.......’` to stop reading input?

```
I love CS3233 Competitive
Programming. i also love
AlGoRiThM
.......you must stop after reading this line as it starts with 7 dots
after the first input block, there will be one loooooooooooong line...
```

2. Suppose that we have one long string `T`. We want to check if another string `P` can be found in `T`. Report all the indices where `P` appears in `T` or report `-1` if `P` cannot be found in `T`. For example, if `T = ‘‘I love CS3233 Competitive Programming. i also love AlGoRiThM’’` and `P = ‘I’`, then the output is only {0} (0-based indexing). If uppercase `‘I’` and lowercase `‘i’` are considered different, then the character `‘i’` at index {39} is not part of the output. If `P = ‘love’`, then the output is {2, 46}. If `P = ‘book’`, then the output is {-1}.

   (a) How to find the first occurrence of a substring in a string (if any)?
       Do we need to implement a string matching algorithm (e.g. Knuth-Morris-Pratt algorithm discussed in Section 6.4, etc) or can we just use library functions?
   (b) How to find the next occurrence(s) of a substring in a string (if any)?

3. Suppose we want to do some simple analysis of the characters in `T` and also to transform each character in `T` into lowercase. The required analysis are: How many digits, vowels `[aeiouAEIOU]`, and consonants (other alphabets that are not vowels) are there in `T`? Can you do all these in $O(n)$ where n is the length of the string `T`?

4. Next, we want to break this one long string `T` into *tokens* (substrings) and store them into an array of strings called `tokens`. For this mini task, the *delimiters* of these tokens are spaces and periods (thus breaking sentences into words). For example, if we *tokenize* the string `T` (in lowercase), we will have these `tokens` = {'i', 'love', 'cs3233', 'competitive', 'programming', 'i', 'also', 'love', 'algorithm'}. Then, we want to sort this array of strings lexicographically[3] and then find the lexicographically smallest string. That is, we have sorted `tokens`: {'algorithm', 'also', 'competitive', 'cs3233', 'i', 'i', 'love', 'love', 'programming'}. Thus, the lexicographically smallest string for this example is 'algorithm'.

   (a) How to tokenize a string?
   (b) How to store the tokens (the shorter strings) in an *array* of strings?
   (c) How to sort an array of strings lexicographically?

5. Now, identify which word appears the most in `T`. In order to answer this query, we need to count the frequency of each word. For `T`, the output is either 'i' or 'love', as both appear twice. Which data structure should be used for this mini task?

6. The given text file has one more line after a line that starts with '.......' but the length of this last line is not constrained. Your task is to count how many characters there are in the last line. How to read a string if its length is not known in advance?

> Tasks and Source code: `ch6_01_basic_string.html/cpp/java`

Profile of Algorithm Inventors

Donald Ervin Knuth (born 1938) is a computer scientist and Professor Emeritus at Stanford University. He is the author of the popular Computer Science book: *"The Art of Computer Programming"*. Knuth has been called the 'father' of the analysis of algorithms. Knuth is also the creator of the TEX, the computer typesetting system used in this book.

James Hiram Morris (born 1941) is a Professor of Computer Science. He is a co-discoverer of the Knuth-Morris-Pratt algorithm for string-search.

Vaughan Ronald Pratt (born 1944) is a Professor Emeritus at Stanford University. He was one of the earliest pioneers in the field of computer science. He has made several contributions to foundational areas such as search algorithms, sorting algorithms, and primality testing. He is also a co-discoverer of the Knuth-Morris-Pratt algorithm for string-search.

Saul B. Needleman and **Christian D. Wunsch** jointly published the string alignment Dynamic Programming algorithm in 1970. Their DP algorithm is discussed in this book.

Temple F. Smith is a Professor in biomedical engineering who helped to develop the Smith-Waterman algorithm developed with Michael Waterman in 1981. The Smith-Waterman algorithm serves as the basis for multi sequence comparisons, identifying the segment with the maximum *local* sequence similarity for identifying similar DNA, RNA, and protein segments.

Michael S. Waterman is a Professor at the University of Southern California. Waterman is one of the founders and current leaders in the area of computational biology. His work has contributed to some of the most widely-used tools in the field. In particular, the Smith-Waterman algorithm (developed with Temple F. Smith) is the basis for many sequence comparison programs.

---

[3]Basically, this is a sort order like the one used in our common dictionary.

6.3 Ad Hoc String Processing Problems

Next, we continue our discussion with something light: The Ad Hoc string processing problems. They are programming contest problems involving strings that require no more than basic programming skills and perhaps some basic string processing skills discussed in Section 6.2 earlier. We only need to read the requirements in the problem description carefully and code the usually short solution. Below, we give a list of such Ad Hoc string processing problems with hints. These programming exercises have been further divided into sub-categories.

- Cipher/Encode/Encrypt/Decode/Decrypt
  It is everyone's wish that their private digital communications are secure. That is, their (string) messages can only be read by the intended recipient(s). Many ciphers have been invented for this purpose and many (of the simpler ones) end up as Ad Hoc programming contest problems, each with its own encoding/decoding rules. There are many such problems in UVa online judge [47]. Thus, we have further split this category into two: the easier versus the harder ones. Try solving some of them, especially those that we classify as **must try ***. It is interesting to learn a bit about *Computer Security/Cryptography* by solving these problems.

- Frequency Counting
  In this group of problems, the contestants are asked to count the frequency of a letter (easy, use Direct Addressing Table) or a word (harder, the solution is either using a balanced Binary Search Tree—like C++ STL `map`/Java `TreeMap`—or Hash table). Some of these problems are actually related to Cryptography (the previous sub-category).

- Input Parsing
  This group of problems is not for IOI contestants as the IOI syllabus enforces the input of IOI tasks to be formatted as simple as possible. However, there is no such restriction in ICPC. Parsing problems range from the simpler ones that can be dealt with an iterative parser and the more complex ones involving some grammars that requires recursive descent parser or Java String/Pattern class.

- Solvable with Java String/Pattern class (Regular Expression)

  Some (but rare) string processing problems are solvable with one liner[4] code that use `matches(String regex)`, `replaceAll(String regex, String replacement)`, and/or other useful functions of Java `String` class. To be able to do this, one has to master the concept of Regular Expression (Regex). We will not discuss Regex in detail but we will show two usage examples:

  1. In UVa 325 - Identifying Legal Pascal Real Constants, we are asked to decide if the given line of input is a legal Pascal Real constant. Suppose the line is stored in `String s`, then the following one-liner Java code is the required solution:

     `s.matches("[-+]?\\d+(\\.\\d+([eE][-+]?\\d+)?|[eE][-+]?\\d+)")`

  2. In UVa 494 - Kindergarten Counting Game, we are asked to count how many words are there in a given line. Here, a word is defined as a consecutive sequence of letters (upper and/or lower case). Suppose the line is stored in `String s`, then the following one-liner Java code is the required solution:

     `s.replaceAll("[^a-zA-Z]+", " ").trim().split(" ").length`

---

[4]We can solve these problems without Regular Expression, but the code may be longer.

- Output Formatting
  This is another group of problems that is also not for IOI contestants. This time, the output is the problematic one. In an ICPC problem set, such problems are used as 'coding warm up' or the 'time-waster problem' for the contestants. Practice your coding skills by solving these problems *as fast as possible* as such problems can differentiate the penalty time for each team.

- String Comparison
  In this group of problems, the contestants are asked to compare strings with various criteria. This sub-category is similar to the string matching problems in the next section, but these problems mostly use `strcmp`-related functions.

- Just Ad Hoc
  These are other Ad Hoc string related problems that cannot be classified as one of the other sub categories above.

Programming Exercises related to Ad Hoc String Processing:

- Cipher/Encode/Encrypt/Decode/Decrypt, Easier
  1. UVa 00245 - Uncompress (use the given algorithm)
  2. UVa 00306 - Cipher (can be made faster by avoiding cycle)
  3. UVa 00444 - Encoder and Decoder (each char is mapped to 2 or 3 digits)
  4. UVa 00458 - The Decoder (shift each character's ASCII value by -7)
  5. UVa 00483 - Word Scramble (read char by char from left to right)
  6. UVa 00492 - Pig Latin (ad hoc, similar to UVa 483)
  7. UVa 00641 - Do the Untwist (reverse the given formula and simulate)
  8. UVa 00739 - Soundex Indexing (straightforward conversion problem)
  9. UVa 00795 - Sandorf's Cipher (prepare an 'inverse mapper')
  10. UVa 00865 - Substitution Cypher (simple character substitution mapping)
  11. UVa 10019 - Funny Encryption Method (not hard, find the pattern)
  12. UVa 10222 - Decode the Mad Man (simple decoding mechanism)
  13. **UVa 10851 - 2D Hieroglyphs ... *** (ignore border; treat '\/' as 1/0; read from bottom)
  14. **UVa 10878 - Decode the Tape *** (treat space/'o' as 0/1, then it is binary to decimal conversion)
  15. UVa 10896 - Known Plaintext Attack (try all possible keys; use tokenizer)
  16. UVa 10921 - Find the Telephone (simple conversion problem)
  17. UVa 11220 - Decoding the message (follow instruction in the problem)
  18. **UVa 11278 - One-Handed Typist *** (map QWERTY keys to DVORAK)
  19. UVa 11541 - Decoding (read char by char and simulate)
  20. UVa 11716 - Digital Fortress (simple cipher)
  21. UVa 11787 - Numeral Hieroglyphs (follow the description)
  22. UVa 11946 - Code Number (ad hoc)

  - Cipher/Encode/Encrypt/Decode/Decrypt, Harder
    1. UVa 00213 - Message Decoding (decrypt the message)
    2. UVa 00468 - Key to Success (letter frequency mapping)
    3. ***UVa 00554 - Caesar Cypher **** (try all shifts; output formatting)

4. *UVa 00632 - Compression (II)* (simulate the process, use sorting)
5. *UVa 00726 - Decode* (frequency cypher)
6. UVa 00740 - Baudot Data ... (just simulate the process)
7. UVa 00741 - Burrows Wheeler Decoder (simulate the process)
8. UVa 00850 - Crypt Kicker II (plaintext attack, tricky test cases)
9. UVa 00856 - The Vigenère Cipher (3 nested loops; one for each digit)
10. **UVa 11385 - Da Vinci Code *** (string manipulation + Fibonacci)
11. **UVa 11697 - Playfair Cipher *** (follow the description, a bit tedious)

- Frequency Counting

  1. UVa 00499 - What's The Frequency ... (use 1D array for frequency counting)
  2. UVa 00895 - Word Problem (get the letter frequency of each word, compare with puzzle line)
  3. **UVa 00902 - Password Search *** (read char by char; count word freq)
  4. UVa 10008 - What's Cryptanalysis? (character frequency count)
  5. UVa 10062 - Tell me the frequencies (ASCII character frequency count)
  6. **UVa 10252 - Common Permutation *** (count freq of each alphabet)
  7. UVa 10293 - Word Length and Frequency (straightforward)
  8. UVa 10374 - Election (use `map` for frequency counting)
  9. UVa 10420 - List of Conquests (word frequency counting, use `map`)
  10. UVa 10625 - GNU = GNU'sNotUnix (frequency addition n times)
  11. UVa 10789 - Prime Frequency (check if a letter's frequency is prime)
  12. **UVa 11203 - Can you decide it ... *** (problem description is convoluted, but this problem is actually easy)
  13. UVa 11577 - Letter Frequency (straightforward problem)

- Input Parsing (Non Recursive)

  1. UVa 00271 - Simply Syntax (grammar check, linear scan)
  2. UVa 00327 - Evaluating Simple C ... (implementation can be tricky)
  3. UVa 00391 - Mark-up (use flags, tedious parsing)
  4. UVa 00397 - Equation Elation (iteratively perform the next operation)
  5. UVa 00442 - Matrix Chain Multiplication (properties of matrix chain mult)
  6. UVa 00486 - English-Number Translator (parsing)
  7. UVa 00537 - Artificial Intelligence? (simple formula; parsing is difficult)
  8. UVa 01200 - A DP Problem (LA 2972, Tehran03, tokenize linear equation)
  9. ***UVa 10906 - Strange Integration **** (BNF parsing, iterative solution)
  10. UVa 11148 - Moliu Fractions (extract integers, simple/mixed fractions from a line; a bit of gcd—see Section 5.5.2)
  11. ***UVa 11357 - Ensuring Truth **** (the problem description looks scary—a SAT (satisfiability) problem; the presence of BNF grammar makes one think of recursive descent parser; however, only one clause needs to be satisfied to get TRUE; a clause can be satisfied if for all variables in the clause, its inverse is not in the clause too; now, we have a much simpler problem)
  12. **UVa 11878 - Homework Checker *** (mathematical expression parsing)
  13. *UVa 12543 - Longest Word* (LA6150, HatYai12, iterative parser)

- Input Parsing (Recursive)

  1. UVa 00384 - Slurpys (recursive grammar check)
  2. UVa 00464 - Sentence/Phrase Generator (generate output based on the given BNF grammar)
  3. UVa 00620 - Cellular Structure (recursive grammar check)
  4. **UVa 00622 - Grammar Evaluation *** (recursive BNF grammar check/evaluation)
  5. UVa 00743 - The MTM Machine (recursive grammar check)
  6. **UVa 10854 - Number of Paths *** (recursive parsing plus counting)
  7. *UVa 11070 - The Good Old Times* (recursive grammar evaluation)
  8. ***UVa 11291 - Smeech **** (recursive descent parser)

- Solvable with Java String/Pattern class (Regular Expression)

  1. **UVa 00325 - Identifying Legal ... *** (see the Java solution above)
  2. **UVa 00494 - Kindergarten Counting ... *** (see the Java solution above)
  3. UVa 00576 - Haiku Review (parsing, grammar)
  4. **UVa 10058 - Jimmi's Riddles *** (solvable with Java regular expression)

- Output Formatting

  1. UVa 00110 - Meta-loopless sort (actually an ad hoc sorting problem)
  2. *UVa 00159 - Word Crosses* (tedious output formatting problem)
  3. UVa 00320 - Border (requires flood fill technique)
  4. *UVa 00330 - Inventory Maintenance* (use `map` to help)
  5. *UVa 00338 - Long Multiplication* (tedious)
  6. *UVa 00373 - Romulan Spelling* (check 'g' versus 'p', ad hoc)
  7. *UVa 00426 - Fifth Bank of ...* (tokenize; sort; reformat output)
  8. UVa 00445 - Marvelous Mazes (simulation, output formatting)
  9. **UVa 00488 - Triangle Wave *** (use several loops)
  10. UVa 00490 - Rotating Sentences (2d array manipulation, output formatting)
  11. *UVa 00570 - Stats* (use `map` to help)
  12. *UVa 00645 - File Mapping* (use recursion to simulate directory structure, it helps the output formatting)
  13. *UVa 00890 - Maze (II)* (simulation, follow the steps, tedious)
  14. UVa 01219 - Team Arrangement (LA 3791, Tehran06)
  15. *UVa 10333 - The Tower of ASCII* (a real time waster problem)
  16. UVa 10500 - Robot maps (simulate, output formatting)
  17. UVa 10761 - Broken Keyboard (tricky with output formatting; note that 'END' is part of input!)
  18. **UVa 10800 - Not That Kind of Graph *** (tedious problem)
  19. *UVa 10875 - Big Math* (simple but tedious problem)
  20. UVa 10894 - Save Hridoy (how fast can you can solve this problem?)
  21. UVa 11074 - Draw Grid (output formatting)
  22. *UVa 11482 - Building a Triangular ...* (tedious...)
  23. UVa 11965 - Extra Spaces (replace consecutive spaces with only one space)
  24. ***UVa 12155 - ASCII Diamondi **** (use proper index manipulation)
  25. *UVa 12364 - In Braille* (2D array check, check all possible digits [0..9])

- String Comparison

    1. UVa 00409 - Excuses, Excuses (tokenize and compare with list of excuses)
    2. **UVa 00644 - Immediate Decodability *** (use brute force)
    3. UVa 00671 - Spell Checker (string comparison)
    4. *UVa 00912 - Live From Mars* (simulation, find and replace)
    5. **UVa 11048 - Automatic Correction ... *** (flexible string comparison with respect to a dictionary)
    6. **UVa 11056 - Formula 1 *** (sorting, case-insensitive string comparison)
    7. UVa 11233 - Deli Deli (string comparison)
    8. UVa 11713 - Abstract Names (modified string comparison)
    9. UVa 11734 - Big Number of Teams ... (modified string comparison)

- Just Ad Hoc

    1. UVa 00153 - Permalex (find formula for this, similar to UVa 941)
    2. UVa 00263 - Number Chains (sort digits, convert to integers, check cycle)
    3. UVa 00892 - Finding words (basic string processing problem)
    4. **UVa 00941 - Permutations *** (formula to get the n-th permutation)
    5. UVa 01215 - String Cutting (LA 3669, Hanoi06)
    6. UVa 01239 - Greatest K-Palindrome ... (LA 4144, Jakarta08, brute-force)
    7. UVa 10115 - Automatic Editing (simply do what they want, uses string)
    8. *UVa 10126 - Zipf's Law* (sort the words to simplify this problem)
    9. UVa 10197 - Learning Portuguese (must follow the description very closely)
    10. UVa 10361 - Automatic Poetry (read, tokenize, process as requested)
    11. UVa 10391 - Compound Words (more like data structure problem)
    12. **UVa 10393 - The One-Handed Typist *** (follow problem description)
    13. UVa 10508 - Word Morphing (number of words = number of letters + 1)
    14. UVa 10679 - I Love Strings (the test data weak; just checking if T is a prefix of S is AC when it should not)
    15. **UVa 11452 - Dancing the Cheeky ... *** (string period, small input, BF)
    16. UVa 11483 - Code Creator (straightforward, use 'escape character')
    17. UVa 11839 - Optical Reader (illegal if mark 0 or > 1 alternatives)
    18. UVa 11962 - DNA II (find formula; similar to UVa 941; base 4)
    19. *UVa 12243 - Flowers Flourish ...* (simple string tokenizer problem)
    20. *UVa 12414 - Calculating Yuan Fen* (brute force problem involving string)

6.4   String Matching

String *Matching* (a.k.a String *Searching*[5]) is a problem of finding the starting index (or indices) of a (sub)string (called *pattern* P) in a longer string (called *text* T). Example: Let's assume that we have T = 'STEVEN EVENT'. If P = 'EVE', then the answers are index 2 and 7 (0-based indexing). If P = 'EVENT', then the answer is index 7 only. If P = 'EVENING', then there is no answer (no matching found and usually we return either -1 or NULL).

6.4.1   Library Solutions

For most *pure* String Matching problems on reasonably short strings, we can just use string library in our programming language. It is strstr in C <string.h>, find in C++ <string>, indexOf in Java String class. Please revisit Section 6.2, mini task 2 that discusses these string library solutions.

6.4.2   Knuth-Morris-Pratt's (KMP) Algorithm

In Section 1.2.3, Question 7, we have an exercise of finding all the occurrences of a substring P (of length m) in a (long) string T (of length n), if any. The code snippet, reproduced below with comments, is actually the *naïve* implementation of String Matching algorithm.

```
void naiveMatching() {
  for (int i = 0; i < n; i++) {        // try all potential starting indices
    bool found = true;
    for (int j = 0; j < m && found; j++)        // use boolean flag 'found'
      if (i + j >= n || P[j] != T[i + j])              // if mismatch found
        found = false;      // abort this, shift the starting index i by +1
    if (found)                                  // if P[0..m-1] == T[i..i+m-1]
      printf("P is found at index %d in T\n", i);
} }
```

This naïve algorithm can run in $O(n)$ *on average* if applied to natural text like the paragraphs of this book, but it can run in $O(nm)$ with the worst case programming contest input like this: T = 'AAAAAAAAAAB' ('A' ten times and then one 'B') and P = 'AAAAB'. The naive algorithm will keep failing at the last character of pattern P and then try the next starting index which is just +1 than the previous attempt. This is not efficient. Unfortunately, a good problem author will include such test case in their secret test data.

In 1977, Knuth, Morris, and Pratt—thus the name of KMP—invented a better String Matching algorithm that makes use of the information gained by previous character comparisons, especially those that matches. KMP algorithm *never* re-compares a character in T that has matched a character in P. However, it works similar to the naïve algorithm if the *first* character of pattern P and the current character in T is a mismatch. In the example below[6], comparing P[j] and T[i] and from i = 0 to 13 with j = 0 (the first character of P) is no different than the naïve algorithm.

---

[5]We deal with this String Matching problem almost every time we read/edit text using computer. How many times have you pressed the well-known 'CTRL + F' button (standard Windows shortcut for the 'find feature') in typical word processing softwares, web browsers, etc?

[6]The sentence in string T below is just for illustration. It is not grammatically correct.

```
                1           2           3           4           5
        01234567890123456789012345678901234567890123456789 0
T = I DO NOT LIKE SEVENTY SEV BUT SEVENTY SEVENTY SEVEN
P = SEVENTY SEVEN
        0123456789012
                1
        ^ the first character of P mismatch with T[i] from index i = 0 to 13
        KMP has to shift the starting index i by +1, as with naive matching.
... at i = 14 and j = 0 ...
                1           2           3           4           5
        01234567890123456789012345678901234567890123456789 0
T = I DO NOT LIKE SEVENTY SEV BUT SEVENTY SEVENTY SEVEN
P =             SEVENTY SEVEN
                0123456789012
                        1
                        ^ then mismatch at index i = 25 and j = 11
```

There are 11 matches from index i = 14 to 24, but one mismatch at i = 25 (j = 11). The naïve matching algorithm will inefficiently restart from index i = 15 but KMP can resume from i = 25. This is because the matched characters before the mismatch is 'SEVENTY SEV'. 'SEV' (of length 3) appears as BOTH proper suffix and prefix of '<u>SEV</u>ENTY <u>SEV</u>'. This 'SEV' is also called as the **border** of 'SEVENTY SEV'. We can safely skip index i = 14 to 21: 'SEVENTY ' in '<u>SEVENTY </u>SEV' as it will not match again, but we cannot rule out the possibility that the next match starts from the second 'SEV'. So, KMP resets j back to 3, skipping 11 - 3 = 8 characters of 'SEVENTY ' (notice the trailing space), while i remains at index 25. This is the major difference between KMP and the naïve matching algorithm.

```
... at i = 25 and j = 3 (This makes KMP efficient) ...
                1           2           3           4           5
        01234567890123456789012345678901234567890123456789 0
T = I DO NOT LIKE SEVENTY SEV BUT SEVENTY SEVENTY SEVEN
P =                     SEVENTY SEVEN
                        0123456789012
                             1
                        ^ then immediate mismatch at index i = 25, j = 3
```

This time the prefix of P before mismatch is 'SEV', but it does not have a border, so KMP resets j back to 0 (or in another word, restart matching pattern P from the front again).

```
... mismatches from i = 25 to i = 29... then matches from i = 30 to i = 42 ...
                1           2           3           4           5
        01234567890123456789012345678901234567890123456789 0
T = I DO NOT LIKE SEVENTY SEV BUT SEVENTY SEVENTY SEVEN
P =                         SEVENTY SEVEN
                            0123456789012
                                    1
```

This is a match, so P = 'SEVENTY SEVEN' is found at index i = 30. After this, KMP knows that '<u>SEVEN</u>TY <u>SEVEN</u>' has 'SEVEN' (of length 5) as border, so KMP resets j back to 5, effectively skipping 13 - 5 = 8 characters of 'SEVENTY ' (notice the trailing space), immediately resumes the search from i = 43, and gets another match. This is efficient.

```
... at i = 43 and j = 5, we have matches from i = 43 to i = 50 ...
So P = 'SEVENTY SEVEN' is found again at index i = 38.
              1         2         3         4         5
     01234567890123456789012345678901234567890123456789 0
T = I DO NOT LIKE SEVENTY SEV BUT SEVENTY SEVENTY SEVEN
P =                                   SEVENTY SEVEN
                                      0123456789012
                                                1
```

To achieve such speed up, KMP has to preprocess the pattern string and get the 'reset table'
b (back). If given pattern string P = 'SEVENTY SEVEN', then table b will looks like this:

```
                         1
     0 1 2 3 4 5 6 7 8 9 0 1 2 3
P =  S E V E N T Y   S E V E N
b = -1 0 0 0 0 0 0 0 0 1 2 3 4 5
```

This means, if mismatch happens in j = 11 (see the example above), i.e. after finding
matches for 'SEVENTY SEV', then we know that we have to re-try matching P from index
j = b[11] = 3, i.e. KMP now assumes that it has matched only the first three characters of
'SEVENTY SEV', which is 'SEV', because the next match can start with that prefix 'SEV'.
The relatively short implementation of the KMP algorithm with comments is shown below.
This implementation has a time complexity of $O(n + m)$.

```
#define MAX_N 100010
char T[MAX_N], P[MAX_N];                          // T = text, P = pattern
int b[MAX_N], n, m;     // b = back table, n = length of T, m = length of P

void kmpPreprocess() {              // call this before calling kmpSearch()
  int i = 0, j = -1; b[0] = -1;                          // starting values
  while (i < m) {                         // pre-process the pattern string P
    while (j >= 0 && P[i] != P[j]) j = b[j];  // different, reset j using b
    i++; j++;                              // if same, advance both pointers
    b[i] = j; // observe i = 8, 9, 10, 11, 12, 13 with j = 0, 1, 2, 3, 4, 5
} }                       // in the example of P = "SEVENTY SEVEN" above

void kmpSearch() {   // this is similar as kmpPreprocess(), but on string T
  int i = 0, j = 0;                                      // starting values
  while (i < n) {                             // search through string T
    while (j >= 0 && T[i] != P[j]) j = b[j];  // different, reset j using b
    i++; j++;                              // if same, advance both pointers
    if (j == m) {                             // a match found when j == m
      printf("P is found at index %d in T\n", i - j);
      j = b[j];                        // prepare j for the next possible match
} } }
```

Source code: `ch6_02_kmp.cpp/java`

Exercise 6.4.1*: Run `kmpPreprocess()` on P = 'ABABA' and show the reset table b!

Exercise 6.4.2*: Run `kmpSearch()` with P = 'ABABA' and T = 'ACABAABABDABABA'.
Explain how the KMP search looks like?

6.4.3 String Matching in a 2D Grid

The string matching problem can also be posed in 2D. Given a 2D grid/array of characters (instead of the well-known 1D array of characters), find the occurrence(s) of pattern P in the grid. Depending on the problem requirement, the search direction can be to 4 or 8 cardinal directions, and either the pattern must be found in a straight line or it can bend. See the following example below.

```
abcdefghigg        // From UVa 10010 - Where's Waldorf?
hebkWaldork        // We can go to 8 directions, but must be straight
ftyawAldorm        // 'WALDORF' is highlighted as capital letters in the grid
ftsimrLqsrc
byoarbeDeyv        // Can you find 'BAMBI' and 'BETTY'?
klcbqwikOmk
strebgadhRb        // Can you find 'DAGBERT' in this row?
yuiqlxcnbjF
```

The solution for such string matching in a 2D grid is usually a *recursive backtracking* (see Section 3.2.2). This is because unlike the 1D counterpart where we always go to the right, at every coordinate (row, col) of the 2D grid, we have *more than one choice* to explore.

To speed up the backtracking process, usually we employ this simple pruning strategy: Once the recursion depth exceeds the length of pattern P, we can immediately prune that recursive branch. This is also called as *depth-limited search* (see Section 8.2.5).

Programming Exercises related to String Matching

- Standard
    1. UVa 00455 - Periodic String (find s in s + s)
    2. *UVa 00886 - Named Extension Dialing* (convert first letter of given name and all the letters of the surname into digits; then do a kind of special string matching where we want the matching to start at the *prefix* of a string)
    3. **UVa 10298 - Power Strings *** (find s in s + s, similar to UVa 455)
    4. UVa 11362 - Phone List (string sort, matching)
    5. **UVa 11475 - Extend to Palindromes *** ('border' of KMP)
    6. **UVa 11576 - Scrolling Sign *** (modified string matching; complete search)
    7. UVa 11888 - Abnormal 89's (to check 'alindrome', find reverse of s in s + s)
    8. *UVa 12467 - Secret word* (similar idea with UVa 11475, if you can solve that problem, you should be able to solve this problem)
- In 2D Grid
    1. **UVa 00422 - Word Search Wonder *** (2D grid, backtracking)
    2. *UVa 00604 - The Boggle Game* (2D matrix, backtracking, sort, and compare)
    3. *UVa 00736 - Lost in Space* (2D grid, a bit modified)
    4. **UVa 10010 - Where's Waldorf? *** (discussed in this section)
    5. **UVa 11283 - Playing Boggle *** (2D grid, backtracking, do not count twice)

6.5 String Processing with Dynamic Programming

In this section, we discuss several string processing problems that are solvable with DP technique discussed in Section 3.5. The first two (String Alignment and Longest Common Subsequence) are *classical* problems and should be known by all competitive programmers. Additionally, we have added a collection of some known twists of these problems.

An important note: For various DP problems on string, we usually manipulate the *integer indices* of the strings and not the actual strings (or substrings) themselves. Passing substrings as parameters of recursive functions is strongly not recommended as it is very slow and hard to memoize.

6.5.1 String Alignment (Edit Distance)

The String Alignment (or Edit Distance[7]) problem is defined as follows: Align[8] two strings `A` with `B` with the maximum alignment score (or minimum number of edit operations):

After aligning `A` with `B`, there are a few possibilities between character `A[i]` and `B[i]`:
1. Character `A[i]` and `B[i]` **match** and we do nothing (assume this worth '+2' score),
2. Character `A[i]` and `B[i]` **mismatch** and we replace `A[i]` with `B[i]` (assume '-1' score),
3. We insert a space in `A[i]` (also '-1' score),
4. We delete a letter from `A[i]` (also '-1' score).

For example: (note that we use a special symbol '_' to denote a space)

```
A = 'ACAATCC' -> 'A_CAATCC'        // Example of a non optimal alignment
B = 'AGCATGC' -> 'AGCATGC_'             // Check the optimal one below
               2-22--2-            // Alignment Score = 4*2 + 4*-1 = 4
```

A brute force solution that tries all possible alignments will get TLE even for medium-length strings `A` and/or `B`. The solution for this problem is the Needleman-Wunsch's (bottom-up) DP algorithm [62]. Consider two strings `A[1..n]` and `B[1..m]`. We define $V(i, j)$ to be the score of the optimal alignment between prefix `A[1..i]` and `B[1..j]` and $score(C1, C2)$ is a function that returns the score if character $C1$ is aligned with character $C2$.

Base cases:
$V(0, 0) = 0$ // no score for matching two empty strings
$V(i, 0) = i \times score(A[i], \_)$ // delete substring `A[1..i]` to make the alignment, $i > 0$
$V(0, j) = j \times score(\_, B[j])$ // insert spaces in `B[1..j]` to make the alignment, $j > 0$

Recurrences: For $i > 0$ and $j > 0$:
$V(i, j) = max(option1, option2, option3)$, where
$option1 = V(i - 1, j - 1) + score(A[i], B[j])$ // score of match or mismatch
$option2 = V(i - 1, j) + score(A[i], \_)$ // delete A_i
$option3 = V(i, j - 1) + score(\_, B[j])$ // insert B_j

In short, this DP algorithm concentrates on the three possibilities for the last pair of characters, which must be either a match/mismatch, a deletion, or an insertion. Although we do not know which one is the best, we can try all possibilities while avoiding the re-computation of overlapping subproblems (i.e. basically a DP technique).

---

[7]Another name for 'edit distance' is 'Levenshtein Distance'. One notable application of this algorithm is the spelling checker feature commonly found in popular text editors. If a user misspells a word, like 'probelm', then a clever text editor that realizes that this word has a very close edit distance to the correct word 'problem' can do the correction automatically.

[8]Align is a process of inserting spaces to strings A or B such that they have the same number of characters. You can view 'inserting spaces to B' as 'deleting the corresponding aligned characters of A'.

```
A = 'xxx...xx'        A = 'xxx...xx'        A = 'xxx...x_'
         |                     |                     |
B = 'yyy...yy'        B = 'yyy...y_'        B = 'yyy...yy'
match/mismatch        delete                insert
```

	_	A	G	C	A	T	G	C
_	0	-1	-2	-3	-4	-5	-6	-7
A	-1							
C	-2		Base Cases					
A	-3							
A	-4							
T	-5							
C	-6							
C	-7							

	_	A	G	C	A	T	G	C
_	0	-1	-2	-3	-4	-5	-6	-7
A	-1	2	1	0	-1	-2	-3	-4
C	-2	1	1	3				
A	-3							
A	-4							
T	-5							
C	-6							
C	-7							

	_	A	G	C	A	T	G	C
_	0	-1	-2	-3	-4	-5	-6	-7
A	-1	2	1	0	-1	-2	-3	-4
C	-2	1	1	3	2	1	0	-1
A	-3	0	0	2	5	4	3	2
A	-4	-1	-1	1	4	4	3	2
T	-5	-2	-2	0	3	6	5	4
C	-6	-3	-3	0	2	5	5	7
C	-7	-4	-4	-1	1	4	4	7
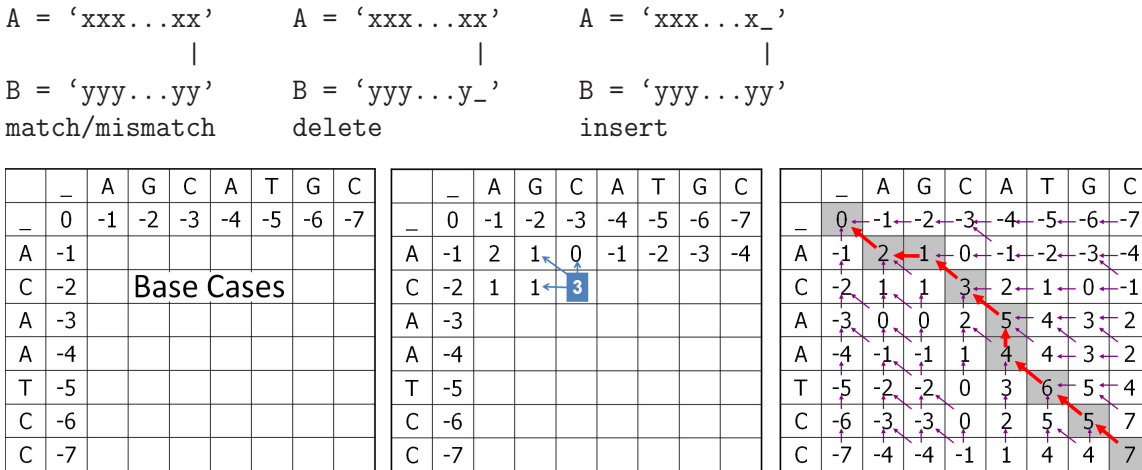
Figure 6.1: Example: `A = 'ACAATCC'` and `B = 'AGCATGC'` (alignment score = 7)

With a simple scoring function where a match gets a +2 points and mismatch, insert, delete all get a -1 point, the detail of string alignment score of `A = 'ACAATCC'` and `B = 'AGCATGC'` is shown in Figure 6.1. Initially, only the base cases are known. Then, we can fill the values row by row, left to right. To fill in $V(i, j)$ for $i, j > 0$, we just need three other values: $V(i - 1, j - 1)$, $V(i - 1, j)$, and $V(i, j - 1)$—see Figure 6.1, middle, row 2, column 3. The maximum alignment score is stored at the bottom right cell (7 in this example).

To reconstruct the solution, we follow the darker cells from the bottom right cell. The solution for the given strings `A` and `B` is shown below. Diagonal arrow means a match or a mismatch (e.g. the last character `..C`). Vertical arrow means a deletion (e.g. `..CAA..` to `..C_A..`). Horizontal arrow means an insertion (e.g. `A_C..` to `AGC..`).

```
A = 'A_CAAT[C]C'                              // Optimal alignment
B = 'AGC_AT[G]C'              // Alignment score = 5*2 + 3*-1 = 7
```

The space complexity of this (bottom-up) DP algorithm is $O(nm)$—the size of the DP table. We need to fill in all cells in the table in $O(1)$ per cell. Thus, the time complexity is $O(nm)$.

Source code: `ch6_03_str_align.cpp/java`

Exercise 6.5.1.1: Why is the cost of a match +2 and the costs of replace, insert, delete are all -1? Are they magic numbers? Will +1 for match work? Can the costs for replace, insert, delete be different? Restudy the algorithm and discover the answer.

Exercise 6.5.1.2: The example source code: `ch6_03_str_align.cpp/java` only show the optimal alignment *score*. Modify the given code to actually show the *actual alignment*!

Exercise 6.5.1.3: Show how to use the 'space saving trick' shown in Section 3.5 to improve this Needleman-Wunsch's (bottom-up) DP algorithm! What will be the new space and time complexity of your solution? What is the drawback of using such a formulation?

Exercise 6.5.1.4: The String Alignment problem in this section is called the **global** alignment problem and runs in $O(nm)$. If the given contest problem is limited to d insertions or deletions only, we can have a faster algorithm. Find a simple tweak to the Needleman-Wunsch's algorithm so that it performs at most d insertions or deletions and runs faster!

Exercise 6.5.1.5: Investigate the improvement of Needleman-Wunsch's algorithm (the **Smith-Waterman's** algorithm [62]) to solve the **local** alignment problem!

6.5.2 Longest Common Subsequence

The Longest Common Subsequence (LCS) problem is defined as follows: Given two strings A and B, what is the longest common subsequence between them. For example, `A = 'ACAATCC'` and `B = 'AGCATGC'` have LCS of length 5, i.e. `'ACATC'`.

This LCS problem can be reduced to the String Alignment problem presented earlier, so we can use the same DP algorithm. We set the cost for mismatch as negative infinity (e.g. -1 Billion), cost for insertion and deletion as 0, and the cost for match as 1. This makes the Needleman-Wunsch's algorithm for String Alignment to never consider mismatches.

Exercise 6.5.2.1: What is the LCS of `A = 'apple'` and `B = 'people'`?

Exercise 6.5.2.2: The Hamming distance problem, i.e. finding the number of different characters between two equal-length strings, can be reduced to String Alignment problem. Assign an appropriate cost to match, mismatch, insert, and delete so that we can compute the Hamming distance between two strings using Needleman-Wunsch's algorithm!

Exercise 6.5.2.3: The LCS problem can be solved in $O(n \log k)$ when all characters are distinct, e.g. if you are given two permutations as in UVa 10635. Solve this variant!

6.5.3 Non Classical String Processing with DP

UVa 11151 - Longest Palindrome

A palindrome is a string that can be read the same way in either direction. Some variants of palindrome finding problems are solvable with DP technique, e.g. UVa 11151 - Longest Palindrome: Given a string of up to $n = 1000$ characters, determine the length of the longest palindrome that you can make from it by deleting zero or more characters. Examples:

'ADAM' → 'ADA' (of length 3, delete 'M')
'MADAM' → 'MADAM' (of length 5, delete nothing)
'NEVERODDOREVENING' → 'NEVERODDOREVEN' (of length 14, delete 'ING')
'RACEF1CARFAST' → 'RACECAR' (of length 7, delete 'F1' and 'FAST')

The DP solution: let $len(l, r)$ be the length of the longest palindrome from string `A[l..r]`.

Base cases:
If $(l = r)$, then $len(l, r) = 1$. // odd-length palindrome
If $(l + 1 = r)$, then $len(l, r) = 2$ if $(A[l] = A[r])$, or 1 otherwise. // even-length palindrome

Recurrences:
If $(A[l] = A[r])$, then $len(l, r) = 2 + len(l + 1, r - 1)$. // both corner characters are the same
else $len(l, r) = max(len(l, r - 1), len(l + 1, r))$. // increase left side or decrease right side

This DP solution has time complexity of $O(n^2)$.

Exercise 6.5.3.1*: Can we use the Longest Common Subsequence solution shown in Section 6.5.2 to solve UVa 11151? If we can, how? What is the time complexity?

Exercise 6.5.3.2*: Suppose that we are now interested to find the longest palindrome in a given string with length up to $n = 10000$ characters. This time, we are not allowed to delete any character. What should be the solution?

Programming Exercises related to String Processing with DP:

- Classic

    1. UVa 00164 - String Computer (String Alignment/Edit Distance)
    2. **UVa 00526 - Edit Distance *** (String Alignment/Edit Distance)
    3. UVa 00531 - Compromise (Longest Common Subsequence; print the solution)
    4. UVa 01207 - AGTC (LA 3170, Manila06, classical String Edit problem)
    5. UVa 10066 - The Twin Towers (Longest Common Subsequence problem, but not on 'string')
    6. UVa 10100 - Longest Match (Longest Common Subsequence)
    7. **UVa 10192 - Vacation *** (Longest Common Subsequence)
    8. UVa 10405 - Longest Common ... (Longest Common Subsequence)
    9. **UVa 10635 - Prince and Princess *** (find LCS of two permutations)
    10. UVa 10739 - String to Palindrome (variation of edit distance)

- Non Classic

    1. *UVa 00257 - Palinwords* (standard DP palindrome plus brute force checks)
    2. *UVa 10453 - Make Palindrome* (s: (L, R); t: (L+1, R-1) if S[L] == S[R]; or one plus min of(L + 1, R) or (L, R - 1); also print the required solution)
    3. UVa 10617 - Again Palindrome (manipulate indices, not the actual string)
    4. ***UVa 11022 - String Factoring **** (s: the min weight of substring `[i..j]`)
    5. **UVa 11151 - Longest Palindrome *** (discussed in this section)
    6. **UVa 11258 - String Partition *** (discussed in this section)
    7. *UVa 11552 - Fewest Flops* (`dp(i, c)` = minimum number of chunks after considering the first `i` segments ending with character `c`)

Profile of Algorithm Inventors

Udi Manber is an Israeli computer scientist. He works in Google as one of their vice presidents of engineering. Along with Gene Myers, Manber invented Suffix Array data structure in 1991.

Eugene "Gene" Wimberly Myers, Jr. is an American computer scientist and bioinformatician, who is best known for his development of the BLAST (Basic Local Alignment Search Tool) tool for sequence analysis. His 1990 paper that describes BLAST has received over 24000 citations making it among the most highly cited paper ever. He also invented Suffix Array with Udi Manber.

6.6 Suffix Trie/Tree/Array

Suffix Trie, Suffix Tree, and Suffix Array are efficient and related data structures for strings. We do not discuss this topic in Section 2.4 as these data structures are unique to strings.

6.6.1 Suffix Trie and Applications

The **suffix** i (or the i-th suffix) of a string is a 'special case' of substring that goes from the i-th character of the string up to the *last* character of the string. For example, the 2-th suffix of 'STEVEN' is 'EVEN', the 4-th suffix of 'STEVEN' is 'EN' (0-based indexing).

A **Suffix Trie**[9] of a set of strings S is a tree of all possible suffixes of strings in S. Each edge label represents a character. Each vertex represents a suffix indicated by its path label: A sequence of edge labels from root to that vertex. Each vertex is connected to (some of) the other 26 vertices (assuming that we only use uppercase Latin letters) according to the suffixes of strings in S. The common prefix of two suffixes is shared. Each vertex has two boolean flags. The first/second one is to indicate that there exists a suffix/word in S *terminating* in that vertex, respectively. Example: If we have $S =$ {'CAR', 'CAT', 'RAT'}, we have the following suffixes {'CAR', 'AR', 'R', 'CAT', 'AT', 'T', 'RAT', 'AT', 'T'}. After sorting and removing duplicates, we have: {'AR', 'AT', 'CAR', 'CAT', 'R', 'RAT', 'T'}. Figure 6.2 shows the Suffix Trie with 7 suffix terminating vertices (filled circles) and 3 word terminating vertices (filled circles indicated with label 'In Dictionary').
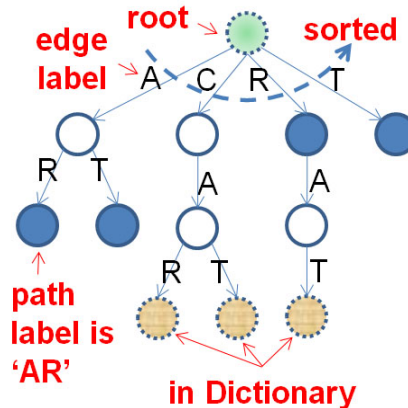
Figure 6.2: Suffix Trie

Suffix Trie is typically used as an efficient data structure for *dictionary*. Assuming that the Suffix Trie of a set of strings in the dictionary has been built, we can determine if a query/pattern string P exists in this dictionary (Suffix Trie) in $O(m)$ where m is the length of string P—this is efficient[10]. We do this by traversing the Suffix Trie from the root. For example, if we want to find if the word $P =$ 'CAT' exists in the Suffix Trie shown in Figure 6.2, we can start from the root node, follow the edge with label 'C', then 'A', then 'T'. Since the vertex at this point has the word-terminating flag set to true, then we know that there is a word 'CAT' in the dictionary. Whereas, if we search for $P =$ 'CAD', we go through this path: root → 'C' → 'A' but then we do not have edge with edge label 'D', so we conclude that 'CAD' is not in the dictionary.

Exercise 6.6.1.1*: Implement this Suffix Trie data structure using the ideas outlined above, i.e. create a vertex object with up to 26 ordered edges that represent 'A' to 'Z' and suffix/word terminating flags. Insert each suffix of each string in S into the Suffix Trie one by one. Analyze the time complexity of such Suffix Trie construction strategy and compare with Suffix Array construction strategy in Section 6.6.4! Also perform $O(m)$ queries for various pattern strings P by starting from the root and follow the corresponding edge labels.

---

[9]This is not a typo. The word 'TRIE' comes from the word 'information reTRIEval'.

[10]Another data structure for dictionary is balanced BST—see Section 2.3. It has $O(\log n \times m)$ performance for each dictionary query where n is the number of words in the dictionary. This is because one string comparison already costs $O(m)$.

6.6.2 Suffix Tree

i	Suffix
0	GATAGACA$
1	ATAGACA$
2	TAGACA$
3	AGACA$
4	GACA$
5	ACA$
6	CA$
7	A$
8	$
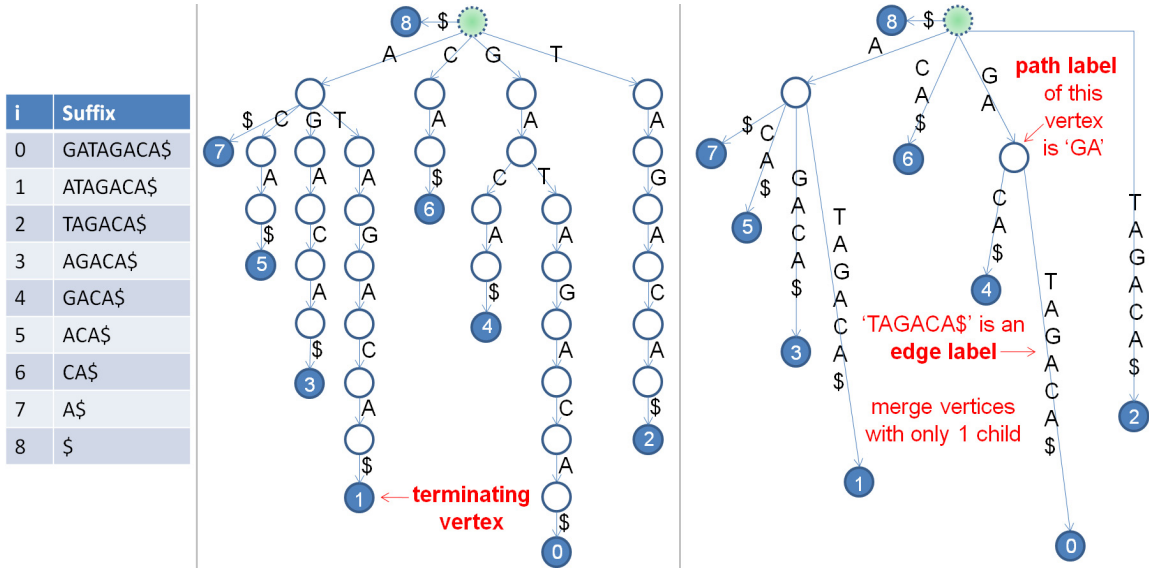
Figure 6.3: Suffixes, Suffix Trie, and Suffix Tree of `T` = '`GATAGACA$`'

Now, instead of working with several short strings, we work with one *long(er)* string. Consider a string `T` = '`GATAGACA$`'. The last character '`$`' is a special terminating character appended to the original string '`GATAGACA`'. It has ASCII value lesser than the characters in `T`. This terminating character ensures that all suffixes terminate in leaf vertices.

The Suffix **Trie** of `T` is shown in Figure 6.3—middle. This time, the **terminating vertex** stores the *index* of the suffix that terminates in that vertex. Observe that the longer the string `T` is, there will be more duplicated vertices in the Suffix Trie. This can be inefficient. Suffix **Tree** of `T` is a Suffix Trie where we *merge* vertices with only one child (essentially a path compression). Compare Figure 6.3—middle and right to see this path compression process. Notice the **edge label** and **path label** in the figure. This time, the edge label can have more than one character. Suffix **Tree** is much more *compact* than Suffix **Trie** with at most $2n$ vertices only[11] (and thus at most $2n-1$ edges). Thus, rather than using Suffix Trie, we will use Suffix Tree in the subsequent sections.

Suffix Tree can be a new data structure for most readers of this book. Therefore in the third edition of this book, we have added a Suffix Tree visualization tool to show the structure of the Suffix Tree of any (but relatively short) input string `T` specified by the reader themselves. Several Suffix Tree applications shown in the next Section 6.6.3 are also included in the visualization.

Visualization: `www.comp.nus.edu.sg/∼stevenha/visualization/suffixtree.html`

Exercise 6.6.2.1*: Draw the Suffix Trie and the Suffix Tree of `T` = '`COMPETITIVE$`'! Hint: Use the Suffix Tree visualization tool shown above.

Exercise 6.6.2.2*: Given two vertices that represents two different suffixes, e.g. suffix 1 and suffix 5 in Figure 6.3—right, determine their Longest Common Prefix! (which is '`A`').

---

[11]There are at most n leaves for n suffixes. All internal non-root vertices are always branching thus there can be at most $n-1$ such vertices. Total: n (leaves) $+ (n-1)$ (internal nodes) $+ 1$ (root) $= 2n$ vertices.

6.6.3 Applications of Suffix Tree

Assuming that the Suffix Tree of a string T is *already built*, we can use it for these applications (not exhaustive):

String Matching in $O(m + occ)$

With Suffix Tree, we can find all (exact) occurrences of a pattern string P in T in $O(m + occ)$ where m is the length of the pattern string P itself and occ is the total number of occurrences of P in T—*no matter how long* the string T is. When the Suffix Tree is *already built*, this approach is *much faster* than string matching algorithms discussed earlier in Section 6.4.

Given the Suffix Tree of T, our task is to search for the vertex x in the Suffix Tree whose path label represents the pattern string P. Remember, a matching is after all a *common prefix* between pattern string P and some suffixes of string T. This is done by just one root to leaf traversal of Suffix Tree of T following the edge labels. Vertex with path label equals to P is the desired vertex x. Then, the suffix indices stored in the terminating vertices (leaves) of the subtree rooted at x are the occurrences of P in T.

Example: In the Suffix Tree of T = 'GATAGACA\$' shown in Figure 6.4 and P = 'A', we can simply traverse from root, go along the edge with edge label 'A' to find vertex x with the path label 'A'. There are 4 occurrences[12] of 'A' in the subtree rooted at x. They are suffix 7: 'A\$', suffix 5: 'ACA\$', suffix 3: 'AGACA\$', and suffix 1: 'ATAGACA\$'.
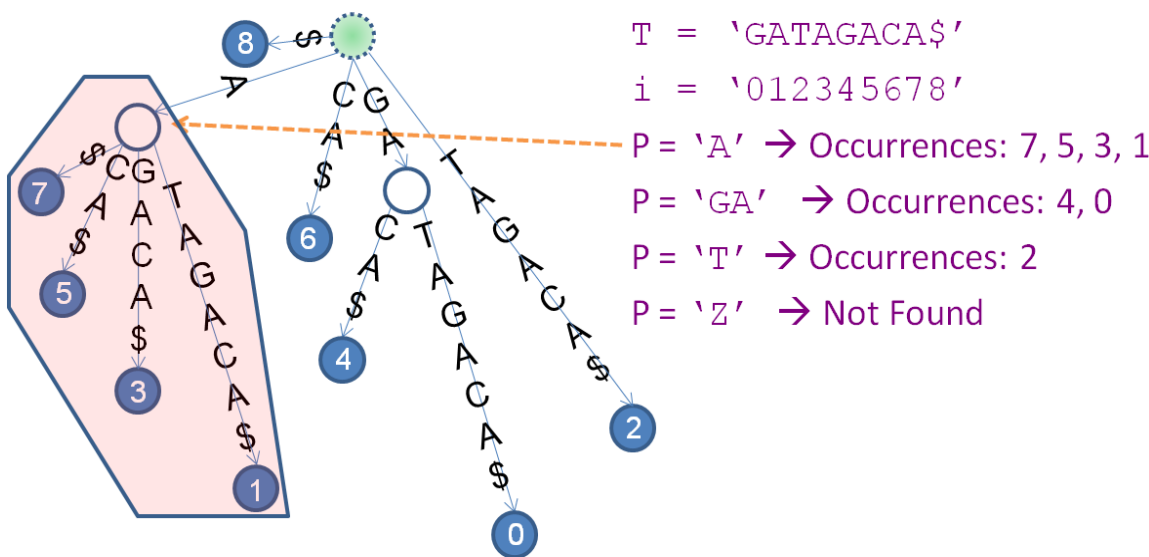
Figure 6.4: String Matching of T = 'GATAGACA\$' with Various Pattern Strings

Finding the Longest Repeated Substring in $O(n)$

Given the Suffix Tree of T, we can also find the Longest Repeated Substring[13] (LRS) in T efficiently. The LRS problem is the problem of finding the longest substring of a string that occurs *at least twice*. The path label of the *deepest internal* vertex x in the Suffix Tree of T is the answer. Vertex x can be found with an $O(n)$ tree traversal. The fact that x is

---

[12] To be precise, occ is the *size* of subtree rooted at x, which can be larger—but not more than double—than the actual number (occ) of terminating vertices (leaves) in the subtree rooted at x.

[13] This problem has several interesting applications: Finding the chorus section of a song (that is repeated several times); Finding the (longest) repeated sentences in a (long) political speech, etc.

an internal vertex implies that it represents more than one suffixes of T (there will be > 1 terminating vertices in the subtree rooted at x) and these suffixes share a common prefix (which implies a repeated substring). The fact that x is the *deepest* internal vertex (from root) implies that its path label is the *longest* repeated substring.

Example: In the Suffix Tree of `T = 'GATAGACA$'` in Figure 6.5, the LRS is `'GA'` as it is the path label of the deepest internal vertex x—`'GA'` is repeated twice in `'GATAGACA$'`.
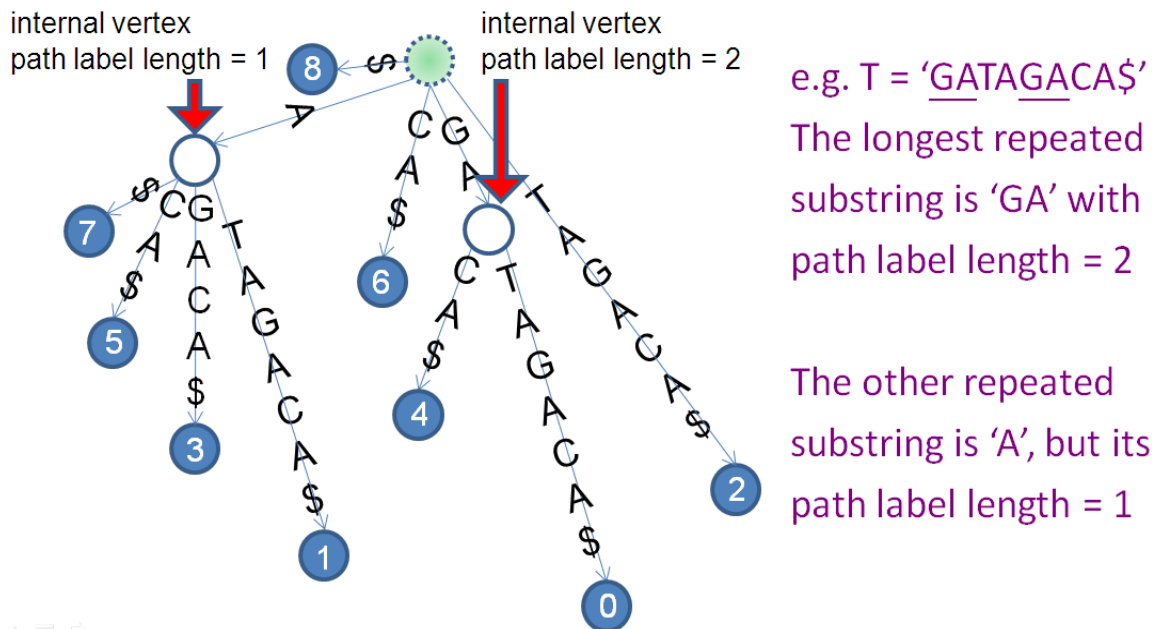
Figure 6.5: Longest Repeated Substring of `T = 'GATAGACA$'`

Finding the Longest Common Substring in $O(n)$

The problem of finding the Longest Common **Substring** (LCS[14]) of two **or more** strings can be solved in linear time[15] with Suffix Tree. Without loss of generality, let's consider the case with *two* strings only: T_1 and T_2. We can build a **generalized Suffix Tree** that combines the Suffix Tree of T_1 and T_2. To differentiate the source of each suffix, we use two different terminating vertex symbols, one for each string. Then, we mark *internal vertices* which have vertices in their subtrees with *different* terminating symbols. The suffixes represented by these marked internal vertices share a common prefix and come from *both* T_1 and T_2. That is, these marked internal vertices represent the common substrings between T_1 and T_2. As we are interested with the *longest* common substring, we report the path label of the *deepest* marked vertex as the answer.

For example, with T_1 = `'GATAGACA$'` and T_2 = `'CATA#'`, The Longest Common Substring is `'ATA'` of length 3. In Figure 6.6, we see the vertices with path labels `'A'`, `'ATA'`, `'CA'`, and `'TA'` have two different terminating symbols (notice that vertex with path label `'GA'` is *not* considered as both suffix `'GACA$'` and `'GATAGACA$'` come from T_1). These are the common substrings between T_1 and T_2. The deepest marked vertex is `'ATA'` and this is the longest common substring between T_1 and T_2.

---

[14]Note that 'Substring' is different from 'Subsequence'. For example, "BCE" is a subsequence but not a substring of "ABCDEF" whereas "BCD" (contiguous) is both a subsequence and a substring of "ABCDEF".

[15]Only if we use the linear time Suffix Tree construction algorithm (not discussed in this book, see [65]).

These are the internal vertices representing suffixes from both strings

The deepest one has path label 'ATA'
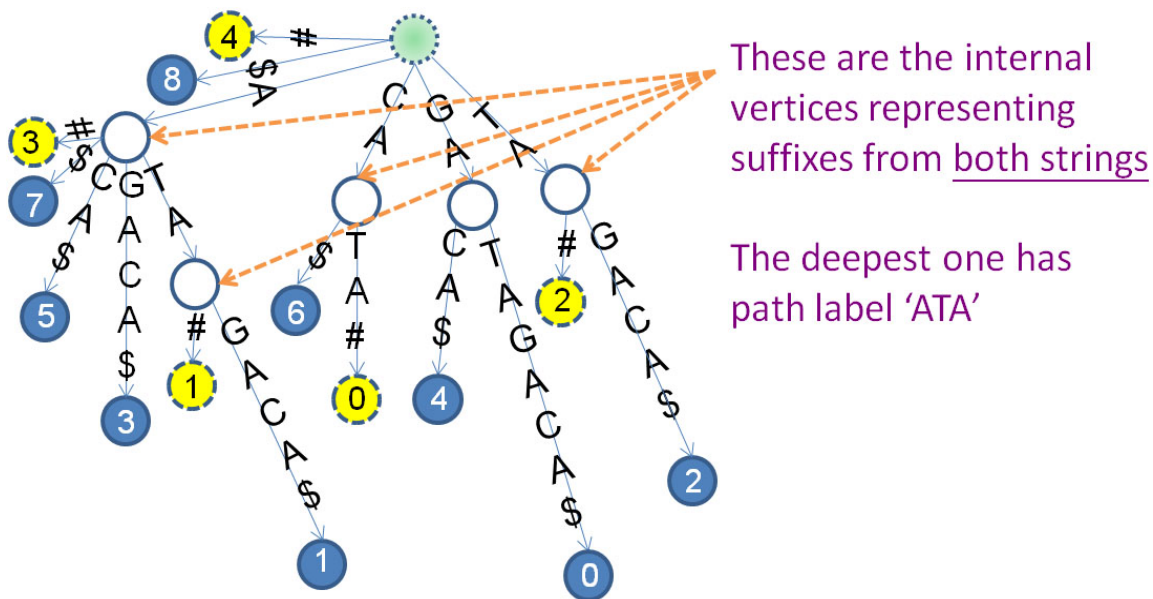
Figure 6.6: Generalized ST of T_1 = 'GATAGACA\$' and T_2 = 'CATA#' and their LCS

Exercise 6.6.3.1: Given the same Suffix Tree in Figure 6.4, find P = 'CA' and P = 'CAT'!

Exercise 6.6.3.2: Find the LRS in T = 'CGACATTACATTA\$'! Build the Suffix Tree first.

Exercise 6.6.3.3*: Instead of finding the LRS, we now want to find the repeated substring *that occurs the most*. Among several possible candidates, pick the longest one. For example, if T = 'DEFG1ABC2DEFG3ABC4ABC\$', the answer is 'ABC' of length 3 that occurs three times (not 'BC' of length 2 or 'C' of length 1 which also occur three times) instead of 'DEFG' of length 4 that occurs only two times. Outline the strategy to find the solution!

Exercise 6.6.3.4: Find the LCS of T_1 = 'STEVEN\$' and T_2 = 'SEVEN#'!

Exercise 6.6.3.5*: Think of how to generalize this approach to find the LCS of *more than two strings*. For example, given three strings T_1 = 'STEVEN\$', T_2 = 'SEVEN#', and T_3 = 'EVE@', how to determine that their LCS is 'EVE'?

Exercise 6.6.3.6*: Customize the solution further so that we find the LCS of *k out of n strings*, where $k \leq n$. For example, given the same three strings T_1, T_2, and T_3 as above, how to determine that the LCS of 2 out of 3 strings is 'EVEN'?

6.6.4 Suffix Array

In the previous subsection, we have shown several string processing problems that can be solved *if the Suffix Tree is already built*. However, the efficient implementation of linear time Suffix Tree construction (see [65]) is complex and thus risky under programming contest setting. Fortunately, the next data structure that we are going to describe—the **Suffix Array** invented by Udi Manber and Gene Myers [43]—has similar functionalities as Suffix Tree but (much) simpler to construct and use, especially in programming contest setting. Thus, we will skip the discussion on $O(n)$ Suffix Tree construction [65] and instead focus on the $O(n \log n)$ Suffix Array construction [68] which is easier to use. Then, in the next subsection, we will show that we can apply Suffix Array to solve problems that have been shown to be solvable with Suffix Tree.

i	Suffix
0	GATAGACA$
1	ATAGACA$
2	TAGACA$
3	AGACA$
4	GACA$
5	ACA$
6	CA$
7	A$
8	$

Sort →

i	SA[i]	Suffix
0	8	$
1	7	A$
2	5	ACA$
3	3	AGACA$
4	1	ATAGACA$
5	6	CA$
6	4	GACA$
7	0	GATAGACA$
8	2	TAGACA$

Figure 6.7: Sorting the Suffixes of T = 'GATAGACA$'

Basically, Suffix Array is an integer array that stores a permutation of n indices of *sorted* suffixes. For example, consider the same T = 'GATAGACA$' with $n = 9$. The Suffix Array of T is a permutation of integers [0..n-1] = {8, 7, 5, 3, 1, 6, 4, 0, 2} as shown in Figure 6.7. That is, the suffixes in sorted order are suffix SA[0] = suffix 8 = '$', suffix SA[1] = suffix 7 = 'A$', suffix SA[2] = suffix 5 = 'ACA$', ..., and finally suffix SA[8] = suffix 2 = 'TAGACA$'.

i	SA[i]	Suffix
0	8	$
1	7	A$
2	5	ACA$
3	3	AGACA$
4	1	ATAGACA$
5	6	CA$
6	4	GACA$
7	0	GATAGACA$
8	2	TAGACA$
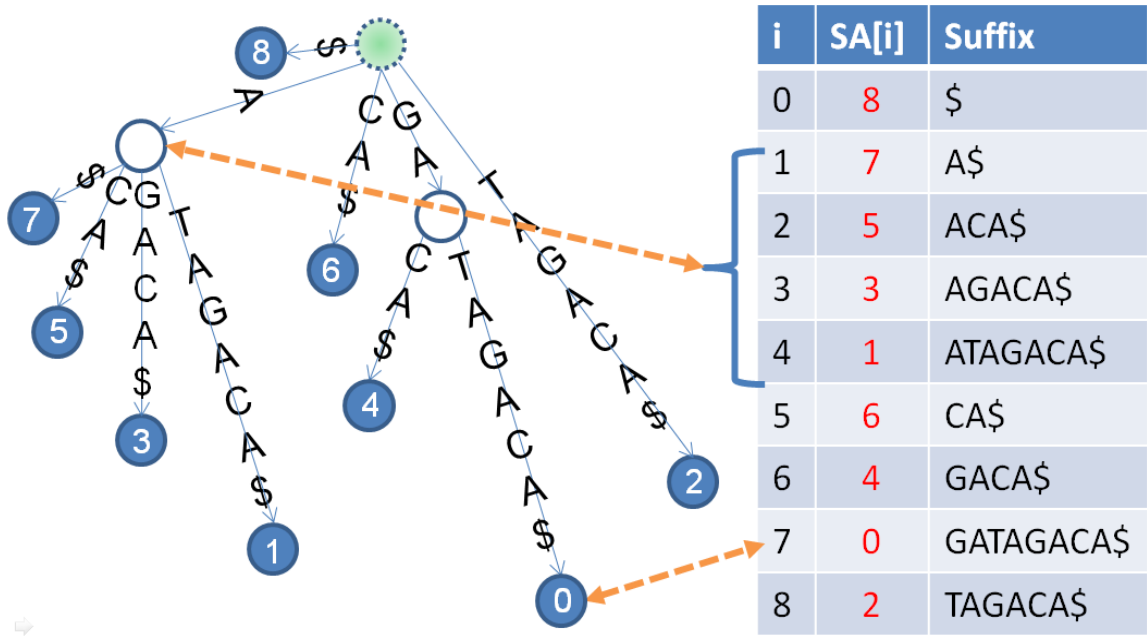
Figure 6.8: Suffix Tree and Suffix Array of T = 'GATAGACA$'

Suffix Tree and Suffix Array are closely related. As we can see in Figure 6.8, the tree traversal of the Suffix Tree visits the terminating vertices (the leaves) in Suffix Array order. An **internal vertex** in Suffix Tree corresponds to a **range** in Suffix Array (a collection of sorted suffixes that share a common prefix). A **terminating vertex** (always at leaf due to the usage of a terminating character) in Suffix Tree corresponds to an **individual index** in Suffix Array (a single suffix). Keep these similarities in mind. They will be useful in the next subsection when we discuss applications of Suffix Array.

Suffix Array is good enough for many challenging string problems involving *long strings* in programming contests. Here, we present two ways to construct a Suffix Array given a string `T[0..n-1]`. The first one is very simple, as shown below:

```
#include <algorithm>
#include <cstdio>
#include <cstring>
using namespace std;

#define MAX_N 1010                              // first approach: O(n^2 log n)
char T[MAX_N];     // this naive SA construction cannot go beyond 1000 chars
int SA[MAX_N], i, n;                           // in programming contest settings

bool cmp(int a, int b) { return strcmp(T + a, T + b) < 0; }        // O(n)

int main() {
  n = (int)strlen(gets(T)); // read line and immediately compute its length
  for (int i = 0; i < n; i++) SA[i] = i; // initial SA: {0, 1, 2, ..., n-1}
  sort(SA, SA + n, cmp);     // sort: O(n log n) * cmp: O(n) = O(n^2 log n)
  for (i = 0; i < n; i++) printf("%2d\t%s\n", SA[i], T + SA[i]);
} // return 0;
```

When applied to string `T = 'GATAGACA$'`, the simple code above that sorts all suffixes with built-in sorting and string comparison *library* produces the correct Suffix Array = {8, 7, 5, 3, 1, 6, 4, 0, 2}. However, this is barely useful except for contest problems with $n \leq 1000$. The overall runtime of this algorithm is $O(n^2 \log n)$ because the `strcmp` operation that is used to determine the order of two (possibly long) suffixes is too costly, up to $O(n)$ per one pair of suffix comparison.

A *better way* to construct Suffix Array is to sort the *ranking pairs* (small integers) of suffixes in $O(\log_2 n)$ iterations from $k = 1, 2, 4, \ldots$, the last **power of 2** that is less than n. At each iteration, this construction algorithm sorts the suffixes based on the ranking pair (`RA[SA[i]]`, `RA[SA[i]+k]`) of suffix `SA[i]`. This algorithm is based on the discussion in [68]. An example execution is shown below for `T = 'GATAGACA$'` and $n = 9$.

- First, `SA[i] = i` and `RA[i]` = ASCII value of `T[i]` $\forall i \in [0..n-1]$ (Table 6.1—left). At iteration $k = 1$, the ranking pair of suffix `SA[i]` is (`RA[SA[i]]`, `RA[SA[i]+1]`).

i	SA[i]	Suffix	RA[SA[i]]	RA[SA[i]+1]	i	SA[i]	Suffix	RA[SA[i]]	RA[SA[i]+1]
0	0	GATAGACA$	71 (G)	65 (A)	0	8	$	36 ($)	00 (-)
1	1	ATAGACA$	65 (A)	84 (T)	1	7	A$	65 (A)	36 ($)
2	2	TAGACA$	84 (T)	65 (A)	2	5	ACA$	65 (A)	67 (C)
3	3	AGACA$	65 (A)	71 (G)	3	3	AGACA$	65 (A)	71 (G)
4	4	GACA$	71 (G)	65 (A)	4	1	ATAGACA$	65 (A)	84 (T)
5	5	ACA$	65 (A)	67 (C)	5	6	CA$	67 (C)	65 (A)
6	6	CA$	67 (C)	65 (A)	6	0	GATAGACA$	71 (G)	65 (A)
7	7	A$	65 (A)	36 ($)	7	4	GACA$	71 (G)	65 (A)
8	8	$	36 ($)	00 (-)	8	2	TAGACA$	84 (T)	65 (A)
Initial ranks RA[i] = ASCII value of T[i]					If SA[i] + k >= n (beyond the length of string T),				
$ = 36, A = 65, C = 67, G = 71, T = 84					we give a default rank 0 with label -				

Table 6.1: L/R: Before/After Sorting; k = 1; the initial sorted order appears

Example 1: The rank of suffix 5 'ACA$' is ('A', 'C') = (65, 67).

Example 2: The rank of suffix 3 'AGACA$' is ('A', 'G') = (65, 71).

After we sort these ranking pairs, the order of suffixes is now like Table 6.1—right, where suffix 5 'ACA$' comes before suffix 3 'AGACA$', etc.

- At iteration $k = 2$, the ranking pair of suffix SA[i] is (RA[SA[i]], RA[SA[i]+2]). This ranking pair is now obtained by looking at the first pair and the second pair of characters only. To get the new ranking pairs, we do not have to recompute many things. We set the first one, i.e. Suffix 8 '$' to have new rank $r = 0$. Then, we iterate from i = [1..n-1]. If the ranking pair of suffix SA[i] is different from the ranking pair of the previous suffix SA[i-1] in sorted order, we increase the rank $r = r + 1$. Otherwise, the rank stays at r (see Table 6.2—left).

i	SA[i]	Suffix	RA[SA[i]]	RA[SA[i]+2]	i	SA[i]	Suffix	RA[SA[i]]	RA[SA[i]+2]
0	8	$	0 ($-)	0 (--)	0	8	$	0 ($-)	0 (--)
1	7	A $	1 (A$)	0 (--)	1	7	A $	1 (A$)	0 (--)
2	5	A C A $	2 (AC)	1 (A$)	2	5	A C A $	2 (AC)	1 (A$)
3	3	A G A C A $	3 (AG)	2 (AC)	3	3	A G A C A $	3 (AG)	2 (AC)
4	1	A T A G A C A $	4 (AT)	3 (AG)	4	1	A T A G A C A $	4 (AT)	3 (AG)
5	6	C A $	5 (CA)	0 ($-)	5	6	C A $	5 (CA)	0 ($-)
6	0	G A T A G A C A $	6 (GA)	7 (TA)	6	4	G A C A $	6 (GA)	5 (CA)
7	4	G A C A $	6 (GA)	5 (CA)	7	0	G A T A G A C A $	6 (GA)	7 (TA)
8	2	T A G A C A $	7 (TA)	6 (GA)	8	2	T A G A C A $	7 (TA)	6 (GA)
$- (first item) is given rank 0, then for i = 1 to n-1, compare rank pair of this row with previous row					If SA[i] + k >= n (beyond the length of string T), we give a default rank 0 with label -				

Table 6.2: L/R: Before/After Sorting; k = 2; 'GATAGACA' and 'GACA' are swapped

Example 1: In Table 6.1—right, the ranking pair of suffix 7 'A$' is (65, 36) which is different with the ranking pair of previous suffix 8 '$-' which is (36, 0). Therefore in Table 6.2—left, suffix 7 has a new rank 1.

Example 2: In Table 6.1—right, the ranking pair of suffix 4 'GACA$' is (71, 65) which is similar with the ranking pair of previous suffix 0 'GATAGACA$' which is also (71, 65). Therefore in Table 6.2—left, since suffix 0 is given a new rank 6, then suffix 4 is also given the same new rank 6.

Once we have updated RA[SA[i]] $\forall i \in$ [0..n-1], the value of RA[SA[i]+k] can be easily determined too. In our explanation, if SA[i]+k $\geq$ n, we give a default rank 0. See **Exercise 6.6.4.2*** for more details on the implementation aspect of this step.

At this stage, the ranking pair of suffix 0 'GATAGACA$' is (6, 7) and suffix 4 'GACA$' is (6, 5). These two suffixes are still not in sorted order whereas all the other suffixes are already in their correct order. After another round of sorting, the order of suffixes is now like Table 6.2—right.

- At iteration $k = 4$, the ranking pair of suffix SA[i] is (RA[SA[i]], RA[SA[i]+4]). This ranking pair is now obtained by looking at the first quadruple and the second quadruple of characters only. Now, notice that the previous ranking pairs of Suffix 4 (6, 5) and Suffix 0 (6, 7) in Table 6.2—right are now different. Therefore, after re-ranking, all n suffixes in Table 6.3 now have different ranking. This can be easily verified by checking if RA[SA[n-1]] == n-1. When this happens, we have successfully obtained the Suffix Array. Notice that the major sorting work is done in the first few iterations only and we usually do not need many iterations.

i	SA[i]	Suffix	RA[SA[i]]	RA[SA[i]+4]
0	8	$	0 ($---)	0 (----)
1	7	A $	1 (A$--)	0 (----)
2	5	A C A $	2 (ACA$)	0 (----)
3	3	A G A C A $	3 (AGAC)	1 (A$--)
4	1	A T A G A C A $	4 (ATAG)	2 (ACA$)
5	6	C A $	5 (CA$-)	0 (----)
6	4	G A C A $	6 (GACA)	0 ($---)
7	0	G A T A G A C A $	7 (GATA)	6 (GACA)
8	2	T A G A C A $	8 (TAGA)	5 (CA$-)

Now all suffixes have different ranking
We are done

Table 6.3: Before/After sorting; k = 4; no change

This Suffix Array construction algorithm can be new for most readers of this book. Therefore in the third edition of this book, we have added a Suffix Array visualization tool to show the steps of of any (but relatively short) input string T specified by the reader themselves. Several Suffix Array applications shown in the next Section 6.6.5 are also included.

Visualization: www.comp.nus.edu.sg/~stevenha/visualization/suffixarray.html

We can implement the sorting of ranking pairs above using (built-in) $O(n \log n)$ sorting library. As we repeat the sorting process up to $\log n$ times, the overall time complexity is $O(\log n \times n \log n) = O(n \log^2 n)$. With this time complexity, we can now work with strings of length up to $\approx 10K$. However, since the sorting process only sort *pair of small integers*, we can use a *linear time* two-pass Radix Sort (that internally calls Counting Sort—see more details in Section 9.32) to reduce the sorting time to $O(n)$. As we repeat the sorting process up to $\log n$ times, the overall time complexity is $O(\log n \times n) = O(n \log n)$. Now, we can work with strings of length up to $\approx 100K$—typical programming contest range.

We provide our $O(n \log n)$ implementation below. Please scrutinize the code to understand how it works. For ICPC contestants only: As you can bring hard copy materials to the contest, it is a good idea to put this code in your team's library.

```
#define MAX_N 100010                             // second approach: O(n log n)
char T[MAX_N];                        // the input string, up to 100K characters
int n;                                         // the length of input string
int RA[MAX_N], tempRA[MAX_N];          // rank array and temporary rank array
int SA[MAX_N], tempSA[MAX_N];       // suffix array and temporary suffix array
int c[MAX_N];                                      // for counting/radix sort

void countingSort(int k) {                                           // O(n)
  int i, sum, maxi = max(300, n);     // up to 255 ASCII chars or length of n
  memset(c, 0, sizeof c);                          // clear frequency table
  for (i = 0; i < n; i++)         // count the frequency of each integer rank
    c[i + k < n ? RA[i + k] : 0]++;
  for (i = sum = 0; i < maxi; i++) {
    int t = c[i]; c[i] = sum; sum += t; }
  for (i = 0; i < n; i++)                 // shuffle the suffix array if necessary
    tempSA[c[SA[i]+k < n ? RA[SA[i]+k] : 0]++] = SA[i];
  for (i = 0; i < n; i++)                          // update the suffix array SA
    SA[i] = tempSA[i];
}
```

```
void constructSA() {          // this version can go up to 100000 characters
  int i, k, r;
  for (i = 0; i < n; i++) RA[i] = T[i];                    // initial rankings
  for (i = 0; i < n; i++) SA[i] = i;     // initial SA: {0, 1, 2, ..., n-1}
  for (k = 1; k < n; k <<= 1) {          // repeat sorting process log n times
    countingSort(k);   // actually radix sort: sort based on the second item
    countingSort(0);           // then (stable) sort based on the first item
    tempRA[SA[0]] = r = 0;                 // re-ranking; start from rank r = 0
    for (i = 1; i < n; i++)                        // compare adjacent suffixes
      tempRA[SA[i]] = // if same pair => same rank r; otherwise, increase r
      (RA[SA[i]] == RA[SA[i-1]] && RA[SA[i]+k] == RA[SA[i-1]+k]) ? r : ++r;
    for (i = 0; i < n; i++)                        // update the rank array RA
      RA[i] = tempRA[i];
    if (RA[SA[n-1]] == n-1) break;               // nice optimization trick
} }

int main() {
  n = (int)strlen(gets(T));        // input T as per normal, without the '$'
  T[n++] = '$';                            // add terminating character
  constructSA();
  for (int i = 0; i < n; i++) printf("%2d\t%s\n", SA[i], T + SA[i]);
} // return 0;
```

Exercise 6.6.4.1*: Show the steps to compute the Suffix Array of T = 'COMPETITIVE$' with $n = 12$! How many sorting iterations that you need to get the Suffix Array? Hint: Use the Suffix Array visualization tool shown above.

Exercise 6.6.4.2*: In the suffix array code shown above, will the following line:

```
    (RA[SA[i]] == RA[SA[i-1]] && RA[SA[i]+k] == RA[SA[i-1]+k]) ? r : ++r;
```

causes index out of bound in some cases?
That is, will SA[i]+k or SA[i-1]+k ever be $\geq$ n and crash the program? Explain!

Exercise 6.6.4.3*: Will the suffix array code shown above works if the input string T contains a space (ASCII value = 32) inside? Hint: The default terminating character used— i.e. '$'—has ASCII value = 36.

6.6.5 Applications of Suffix Array

We have mentioned earlier that Suffix Array is closely related to Suffix Tree. In this subsection, we show that with Suffix Array (which is easier to construct), we can solve the string processing problems shown in Section 6.6.3 that are solvable using Suffix Tree.

String Matching in $O(m \log n)$

After we obtain the Suffix Array of T, we can search for a pattern string P (of length m) in T (of length n) in $O(m \log n)$. This is a factor of $\log n$ times slower than the Suffix Tree version but in practice is quite acceptable. The $O(m \log n)$ complexity comes from the fact that we can do two $O(\log n)$ binary searches on sorted suffixes and do up to $O(m)$ suffix

comparisons[16]. The first/second binary search is to find the lower/upper bound respectively. This lower/upper bound is the the smallest/largest `i` such that the prefix of suffix `SA[i]` matches the pattern string `P`, respectively. All the suffixes between the lower and upper bound are the occurrences of pattern string `P` in `T`. Our implementation is shown below:

```
ii stringMatching() {                           // string matching in O(m log n)
  int lo = 0, hi = n-1, mid = lo;                  // valid matching = [0..n-1]
  while (lo < hi) {                                          // find lower bound
    mid = (lo + hi) / 2;                                  // this is round down
    int res = strncmp(T + SA[mid], P, m);  // try to find P in suffix 'mid'
    if (res >= 0) hi = mid;          // prune upper half (notice the >= sign)
    else          lo = mid + 1;             // prune lower half including mid
  }                                         // observe '=' in "res >= 0" above
  if (strncmp(T + SA[lo], P, m) != 0) return ii(-1, -1);    // if not found
  ii ans; ans.first = lo;
  lo = 0; hi = n - 1; mid = lo;
  while (lo < hi) {               // if lower bound is found, find upper bound
    mid = (lo + hi) / 2;
    int res = strncmp(T + SA[mid], P, m);
    if (res > 0) hi = mid;                              // prune upper half
    else          lo = mid + 1;             // prune lower half including mid
  }                             // (notice the selected branch when res == 0)
  if (strncmp(T + SA[hi], P, m) != 0) hi--;              // special case
  ans.second = hi;
  return ans;
} // return lower/upperbound as first/second item of the pair, respectively

int main() {
  n = (int)strlen(gets(T));         // input T as per normal, without the '$'
  T[n++] = '$';                                  // add terminating character
  constructSA();
  for (int i = 0; i < n; i++) printf("%2d\t%s\n", SA[i], T + SA[i]);

  while (m = (int)strlen(gets(P)), m) {     // stop if P is an empty string
    ii pos = stringMatching();
    if (pos.first != -1 && pos.second != -1) {
      printf("%s found, SA [%d..%d] of %s\n", P, pos.first, pos.second, T);
      printf("They are:\n");
      for (int i = pos.first; i <= pos.second; i++)
        printf("  %s\n", T + SA[i]);
    } else printf("%s is not found in %s\n", P, T);
} } // return 0;
```

A sample execution of this string matching algorithm on the Suffix Array of `T = 'GATAGACA$'` with `P = 'GA'` is shown in Table 6.4 below.

   We start by finding the lower bound. The current range is `i = [0..8]` and thus the middle one is `i = 4`. We compare the first two characters of suffix `SA[4]`, which is '<u>AT</u>AGACA$', with `P = 'GA'`. As `P = 'GA'` is larger, we continue exploring `i = [5..8]`. Next, we compare the first two characters of suffix `SA[6]`, which is '<u>GA</u>CA$', with `P = 'GA'`. It is a match.

---

[16]This is achievable by using the str<u>n</u>cmp function to compare only the first m characters of both suffixes.

As we are currently looking for the *lower* bound, we do not stop here but continue exploring i = [5..6]. P = 'GA' is larger than suffix SA[5], which is 'CA$'. We stop here. Index i = 6 is the lower bound, i.e. suffix SA[6], which is 'GACA$', is the *first* time pattern P = 'GA' appears as a prefix of a suffix in the list of sorted suffixes.

Finding lower bound

i	SA[i]	Suffix
0	8	$
1	7	A$
2	5	ACA$
3	3	AGACA$
4	1	✖ ATAGACA$
5	6	✖ CA$
6	4	GACA$
7	0	GATAGACA$
8	2	TAGACA$

Finding upper bound

i	SA[i]	Suffix
0	8	$
1	7	A$
2	5	ACA$
3	3	AGACA$
4	1	✖ ATAGACA$
5	6	CA$
6	4	GACA$
7	0	GATAGACA$
8	2	✖ TAGACA$

Table 6.4: String Matching using Suffix Array

Next, we search for the upper bound. The first step is the same as above. But at the second step, we have a match between suffix SA[6], which is 'GACA$', with P = 'GA'. Since now we are looking for the *upper* bound, we continue exploring i = [7..8]. We find another match when comparing suffix SA[7], which is 'GATAGACA$', with P = 'GA'. We stop here. This i = 7 is the upper bound in this example, i.e. suffix SA[7], which is 'GATAGACA$', is the *last* time pattern P = 'GA' appears as a prefix of a suffix in the list of sorted suffixes.

Finding the Longest Common Prefix in $O(n)$

Given the Suffix Array of T, we can compute the Longest Common Prefix (LCP) between *consecutive* suffixes in Suffix Array order. By definition, LCP[0] = 0 as suffix SA[0] is the first suffix in Suffix Array order without any other suffix preceding it. For i > 0, LCP[i] = the length of common prefix between suffix SA[i] and suffix SA[i-1]. See Table 6.5— left. We can compute LCP directly by definition by using the code below. However, this approach is slow as it can increase the value of L up to $O(n^2)$ times. This defeats the purpose of building Suffix Array in $O(n \log n)$ time as shown in Section 6.8.

```
void computeLCP_slow() {
  LCP[0] = 0;                                              // default value
  for (int i = 1; i < n; i++) {                // compute LCP by definition
    int L = 0;                                        // always reset L to 0
    while (T[SA[i] + L] == T[SA[i-1] + L]) L++;    // same L-th char, L++
    LCP[i] = L;
} }
```

A better solution using the Permuted Longest-Common-Prefix (PLCP) theorem [37] is described below. The idea is simple: It is *easier* to compute the LCP in the original position order of the suffixes instead of the lexicographic order of the suffixes. In Table 6.5—right, we have the original position order of the suffixes of T = 'GATAGACA$'. Observe that column PLCP[i] forms a pattern: Decrease-by-1 block $(2 \rightarrow 1 \rightarrow 0)$; increase to 1; decrease-by-1 block again $(1 \rightarrow 0)$; increase to 1 again; decrease-by-1 block again $(1 \rightarrow 0)$, etc.

i	SA[i]	LCP[i]	Suffix	i	Phi[i]	PLCP[i]	Suffix
0	8	0	$	0	4	2	GATAGACA$
1	7	0	A$	1	3	1	ATAGACA$
2	5	1	ACA$	2	0	0	TAGACA$
3	3	1	AGACA$	3	5	1	AGACA$
4	1	1	ATAGACA$	4	6	0	GACA$
5	6	0	CA$	5	7	1	ACA$
6	4	0	GACA$	6	1	0	CA$
7	0	2	GATAGACA$	7	8	0	A$
8	2	0	TAGACA$	8	-1	0	$

LCP[7] = PLCP[SA[7]] = PLCP[0] = 2

Phi[SA[3]] = SA[3-1]
Phi[3] = SA[2]
Phi[3] = 5

Table 6.5: Computing the LCP given the SA of `T` = '`GATAGACA$`'

The PLCP theorem says that the total number of increase (and decrease) operations is at most $O(n)$. This pattern and this $O(n)$ guarantee are exploited in the code below.

First, we compute `Phi[i]`, that stores the suffix index of the previous suffix of suffix `SA[i]` in Suffix Array order. By definition, `Phi[SA[0]]` = `-1`, i.e. there is no previous suffix that precede suffix `SA[0]`. Take some time to verify the correctness of column `Phi[i]` in Table 6.5—right. For example, `Phi[SA[3]]` = `SA[3-1]`, so `Phi[3]` = `SA[2]` = `5`.

Now, with `Phi[i]`, we can compute the permuted LCP. The first few steps of this algorithm is elaborated below. When `i` = `0`, we have `Phi[0]` = `4`. This means suffix 0 '`GATAGACA$`' has suffix 4 '`GACA$`' before it in Suffix Array order. The first two characters (`L` = `2`) of these two suffixes match, so `PLCP[0]` = `2`.

When `i` = `1`, we know that *at least* `L-1` = `1` characters can match as the next suffix in position order will have one less starting character than the current suffix. We have `Phi[1]` = `3`. This means suffix 1 '`ATAGACA$`' has suffix 3 '`AGACA$`' before it in Suffix Array order. Observe that these two suffixes indeed have at least 1 character match (that is, we do not start from `L` = `0` as in `computeLCP_slow()` function shown earlier and therefore this is more efficient). As we cannot extend this further, we have `PLCP[1]` = `1`.

We continue this process until `i` = `n-1`, bypassing the case when `Phi[i]` = `-1`. As the PLCP theorem says that `L` will be increased/decreased at most `n` times, this part runs in amortized $O(n)$. Finally, once we have the PLCP array, we can put the permuted LCP back to the correct position. The code is relatively short, as shown below.

```
void computeLCP() {
  int i, L;
  Phi[SA[0]] = -1;                                      // default value
  for (i = 1; i < n; i++)                               // compute Phi in O(n)
    Phi[SA[i]] = SA[i-1];     // remember which suffix is behind this suffix
  for (i = L = 0; i < n; i++) {            // compute Permuted LCP in O(n)
    if (Phi[i] == -1) { PLCP[i] = 0; continue; }            // special case
    while (T[i + L] == T[Phi[i] + L]) L++;        // L increased max n times
    PLCP[i] = L;
    L = max(L-1, 0);                              // L decreased max n times
  }
  for (i = 0; i < n; i++)                               // compute LCP in O(n)
    LCP[i] = PLCP[SA[i]];    // put the permuted LCP to the correct position
}
```

Finding the Longest Repeated Substring in $O(n)$

If we have computed the Suffix Array in $O(n \log n)$ and the LCP between consecutive suffixes in Suffix Array order in $O(n)$, then we can determine the length of the Longest Repeated Substring (LRS) of T in $O(n)$.

  The length of the longest repeated substring is just the highest number in the LCP array. In Table 6.5—left that corresponds to the Suffix Array and the LCP of T = 'GAT<u>A</u><u>G</u><u>A</u>CA\$', the highest number is 2 at index i = 7. The first 2 characters of the corresponding suffix SA[7] (suffix 0) is 'GA'. This is the longest repeated substring in T.

Finding the Longest Common Substring in $O(n)$

i	SA[i]	LCP[i]	Owner	Suffix
0	13	0	2	#
1	8	0	1	\$CATA#
2	12	0	2	A#
3	7	1	1	<u>A</u>\$CATA#
4	5	1	1	<u>A</u>CA\$CATA#
5	3	1	1	<u>A</u>GACA\$CATA#
6	10	1	2	<u>A</u>TA#
7	**1**	**3**	**1**	**<u>ATA</u>GACA\$CATA#**
8	6	0	1	CA\$CATA#
9	9	2	2	<u>CA</u>TA#
10	4	0	1	GACA\$CATA#
11	0	2	1	<u>GA</u>TAGACA\$CATA#
12	11	0	2	TA#
13	2	2	1	<u>TA</u>GACA\$CATA#

Table 6.6: The Suffix Array, LCP, and owner of T = 'GATAGACA\$CATA#'

Without loss of generality, let's consider the case with only *two* strings. We use the same example as in the Suffix Tree section earlier: T_1 = 'GAT<u>A</u><u>G</u><u>A</u>CA\$' and T_2 = 'CATA#'. To solve the LCS problem using Suffix Array, first we have to concatenate both strings (note that the terminating characters of both strings *must be different*) to produce T = 'GATAGACA\$CATA#'. Then, we compute the Suffix and LCP array of T as shown in Figure 6.6.

  Then, we go through consecutive suffixes in $O(n)$. If two consecutive suffixes belong to different owner (can be easily checked[17], for example we can test if suffix SA[i] belongs to T_1 by testing if SA[i] < the length of T_1), we look at the LCP array and see if the maximum LCP found so far can be increased. After one $O(n)$ pass, we will be able to determine the Longest Common Substring. In Figure 6.6, this happens when i = 7, as suffix SA[7] = suffix 1 = 'ATAGACA\$CATA#' (owned by T_1) and its previous suffix SA[6] = suffix 10 = 'ATA#' (owned by T_2) have a common prefix of length 3 which is 'ATA'. This is the LCS.

  We close this section and this chapter by highlighting the availability of our source code. Please spend some time understanding the source code which may not be trivial for those who are new with Suffix Array.

Source code: `ch6_04_sa.cpp/java`

---

[17]With three or more strings, this check will have more 'if statements'.

Exercise 6.6.5.1*: Suggest some possible improvements to the `stringMatching()` function shown in this section!

Exercise 6.6.5.2*: Compare the KMP algorithm shown in Section 6.4 with the string matching feature of Suffix Array. When it is more beneficial to use Suffix Array to deal with string matching and when it is more beneficial to just use KMP or standard string libraries?

Exercise 6.6.5.3*: Solve all exercises on Suffix Tree applications, i.e. **Exercise 6.6.3.1, 2, 3*, 4, 5*, and 6*** using Suffix Array instead!

Programming Exercises related to Suffix Array[18]:

1. UVa 00719 - Glass Beads (min lexicographic rotation[19]; $O(n \log n)$ build SA)
2. **UVa 00760 - DNA Sequencing *** (Longest Common *Substring* of two strings)
3. UVa 01223 - Editor (LA 3901, Seoul07, Longest Repeated Substring (or KMP))
4. *UVa 01254 - Top 10* (LA 4657, Jakarta09, Suffix Array + Segment Tree)
5. **UVa 11107 - Life Forms *** (Longest Common Substring of $> \frac{1}{2}$ of the strings)
6. **UVa 11512 - GATTACA *** (Longest Repeated Substring)
7. SPOJ 6409 - Suffix Array (problem author: Felix Halim)
8. IOI 2008 - Type Printer (DFS traversal of Suffix Trie)

---

[18]You can try solving these problems with Suffix Tree, but you have to learn how to code the Suffix Tree construction algorithm by yourself. The programming problems listed here are solvable with Suffix Array. Also please take note that our sample code uses `gets` for reading the input strings. If you use `scanf(''%s'')` or `getline`, do not forget to adjust the potential DOS/UNIX 'end of line' differences.

[19]This problem can be solved by concatenating the string with itself, build the Suffix Array, then find the first suffix in Suffix Array sorted order that has length greater or equal to n.

6.7 Solution to Non-Starred Exercises

C Solutions for Section 6.2

Exercise 6.2.1:

(a) A string is stored as an array of characters terminated by null, e.g. `char str[30x10+50]`, `line[30+50];`. It is a good practice to declare array size slightly bigger than requirement to avoid "off by one" bug.

(b) To read the input line by line, we use[20] `gets(line);` or `fgets(line, 40, stdin);` in `string.h` (or `cstring`) library. Note that `scanf(''%s'', line)` is not suitable here as it will only read the first word.

(c) We first set `strcpy(str, '''');`, and then we combine the `line`s that we read into a longer string using `strcat(str, line);`. If the current line is not the last one, we append a space to the back of `str` using `strcat(str, '' '');` so that the last word from this line is not accidentally combined with the first word of the next line.

(d) We stop reading the input when `strncmp(line, ''.......'', 7) == 0`. Note that `strncmp` only compares the first n characters.

Exercise 6.2.2:

(a) For finding a substring in a relatively short string (the standard string matching problem), we can just use library function. We can use `p = strstr(str, substr);` The value of `p` will be `NULL` if `substr` is not found in `str`.

(b) If there are multiple copies of `substr` in `str`, we can use `p = strstr(str + pos, substr)`. Initially `pos = 0`, i.e. we search from the first character of `str`. After finding one occurrence of `substr` in `str`, we can call `p = strstr(str + pos, substr)` again where this time `pos` is the index of the current occurrence of `substr` in `str` *plus one* so that we can get the next occurrence. We repeat this process until `p == NULL`. This C solution requires understanding of the memory address of a C array.

Exercise 6.2.3: In many string processing tasks, we are required to iterate through every characters in `str` once. If there are n characters in `str`, then such scan requires $O(n)$. In both C/C++, we can use `tolower(ch)` and `toupper(ch)` in `ctype.h` to convert a character to its lower and uppercase version. There are also `isalpha(ch)`/`isdigit(ch)` to check whether a given character is alphabet `[A-Za-z]`/digit, respectively. To test whether a character is a vowel, one method is to prepare a string `vowel = "aeiou";` and check if the given character is one of the five characters in `vowel`. To check whether a character is a consonant, simply check if it is an alphabet but not a vowel.

Exercise 6.2.4: Combined C and C++ solutions:

(a) One of the easiest ways to tokenize a string is to use `strtok(str, delimiters);` in C.

(b) These tokens can then be stored in a C++ `vector<string> tokens`.

(c) We can use C++ STL `algorithm::sort` to sort `vector<string> tokens`. When needed, we can convert C++ `string` back to C string by using `str.c_str()`.

---

[20]Note: Function `gets` is actually unsafe because it does not perform bounds checking on input size.

Exercise 6.2.5: See the C++ solution.

Exercise 6.2.6: Read the input character by character and count incrementally, look for the presence of '\n' that signals the end of a line. Pre-allocating a fixed-sized buffer is not a good idea as the problem author can set a ridiculously long string to break your code.

C++ Solutions for Section 6.2

Exercise 6.2.1:

(a) We can use class `string`.

(b) We can use `cin.getline()` in `string` library.

(c) We can use the '+' operator directly to concatenate strings.

(d) We can use the '==' operator directly to compare two strings.

Exercise 6.2.2:

(a) We can use function `find` in class `string`.

(b) Same idea as in C language. We can set the offset value in the second parameter of function `find` in class `string`.

Exercise 6.2.3-4: Same solutions as in C language.

Exercise 6.2.5: We can use C++ STL `map<string, int>` to keep track the frequency of each word. Every time we encounter a new token (which is a string), we increase the corresponding frequency of that token by one. Finally, we scan through all tokens and determine the one with the highest frequency.

Exercise 6.2.6: Same solution as in C language.

Java Solutions for Section 6.2

Exercise 6.2.1:

(a) We can use class `String`, `StringBuffer`, or `StringBuilder` (this one is faster than `StringBuffer`).

(b) We can use the `nextLine` method in Java `Scanner`. For faster I/O, we can consider using the `readLine` method in Java `BufferedReader`.

(c) We can use the `append` method in `StringBuilder`. We should not concatenate Java Strings with the '+' operator as Java String class is immutable and thus such operation is (very) costly.

(d) We can use the `equals` method in Java `String`.

Exercise 6.2.2:

(a) We can use the `indexOf` method in class `String`.

(b) Same idea as in C language. We can set the offset value in the second parameter of `indexOf` method in class `String`.

Exercise 6.2.3: Use Java `StringBuilder` and `Character` classes for these operations.

Exercise 6.2.4:

(a) We can use Java `StringTokenizer` class or the `split` method in Java `String` class.

(b) We can use Java `Vector` of `String`s.

(c) We can use Java `Collections.sort`.

Exercise 6.2.5: Same idea as in C++ language.
We can use Java `TreeMap<String, Integer>`.

Exercise 6.2.6: We need to use the `read` method in Java `BufferedReader` class.

Solutions for the Other Sections

Exercise 6.5.1.1: Different scoring scheme will yield different (global) alignment. If given a string alignment problem, read the problem statement and see what is the required cost for match, mismatch, insert, and delete. Adapt the algorithm accordingly.

Exercise 6.5.1.2: You have to save the predecessor information (the arrows) during the DP computation. Then follow the arrows using recursive backtracking. See Section 3.5.1.

Exercise 6.5.1.3: The DP solution only need to refer to previous row so it can utilize the 'space saving trick' by just using two rows, the current row and the previous row. The new space complexity is just $O(min(n, m))$, that is, put the string with the lesser length as string 2 so that each row has lesser columns (less memory). The time complexity of this solution is still $O(nm)$. The only drawback of this approach, as with any other space saving trick is that we will not be able to reconstruct the optimal solution. So if the actual optimal solution is needed, we cannot use this space saving trick. See Section 3.5.1.

Exercise 6.5.1.4: Simply concentrate along the main diagonal with width d. We can speed up Needleman-Wunsch's algorithm to $O(dn)$ by doing this.

Exercise 6.5.1.5: It involves Kadane's algorithm again (see maximum sum problem in Section 3.5.2).

Exercise 6.5.2.1: 'pple'.

Exercise 6.5.2.2: Set score for match = 0, mismatch = 1, insert and delete = negative infinity. However, this solution is not efficient and not natural, as we can simply use an $O(min(n, m))$ algorithm to scan both string 1 and string 2 and count how many characters are different.

Exercise 6.5.2.3: Reduced to LIS, $O(n \log k)$ solution. The reduction to LIS is not shown. Draw it and see how to reduce this problem into LIS.

Exercise 6.6.3.1: 'CA' is found, 'CAT' is not.

Exercise 6.6.3.2: 'ACATTA'.

Exercise 6.6.3.4: 'EVEN'.

6.8   Chapter Notes

The material about String Alignment (Edit Distance), Longest Common Subsequence, and Suffix Trie/Tree/Array are originally from **A/P Sung Wing Kin, Ken** [62], School of Computing, National University of Singapore. The material have since evolved from a more theoretical style into the current competitive programming style.

The section about basic string processing skills (Section 6.2) and the Ad Hoc string processing problems were born from our experience with string-related problems and techniques. The number of programming exercises mentioned there is about three quarters of all other string processing problems discussed in this chapter. We are aware that these are not the typical ICPC problems/IOI tasks, but they are still good programming exercises to improve your programming skills.

In Section 6.4, we discuss the library solutions and one fast algorithm (Knuth-Morris-Pratt/KMP algorithm) for the String Matching problem. The KMP implementation will be useful if you have to modify basic string matching requirement yet you still need fast performance. We believe KMP is fast enough for finding pattern string in a long string for typical contest problems. Through experimentation, we conclude that the KMP implementation shown in this book is slightly faster than the built-in `C strstr`, `C++ string.find` and `Java String.indexOf`. If an even faster string matching algorithm is needed during contest time for one longer string and much more queries, we suggest using Suffix Array discussed in Section 6.8. There are several other string matching algorithms that are not discussed yet like **Boyer-Moore's**, **Rabin-Karp's**, **Aho-Corasick's**, **Finite State Automata**, etc. Interested readers are welcome to explore them.

We have expanded the discussion of *non classical* DP problems involving string in Section 6.5. We feel that the classical ones will be rarely asked in modern programming contests.

The practical implementation of Suffix Array (Section 6.6) is inspired mainly from the article "Suffix arrays - a programming contest approach" by [68]. We have integrated and synchronized many examples given there with our way of writing Suffix Array implementation. In the third edition of this book, we have (re-)introduced the concept of terminating character in Suffix Tree and Suffix Array as it simplifies the discussion. It is a good idea to solve *all* the programming exercises listed in Section 6.6 although they are not that many *yet*. This is an important data structure that will be more popular in the near future.

Compared to the first two editions of this book, this chapter has grown even more—similar case as with Chapter 5. However, there are several other string processing problems that we have not touched yet: **Hashing Techniques** for solving some string processing problems, the **Shortest Common Superstring** problem, **Burrows-Wheeler transformation** algorithm, **Suffix Automaton**, **Radix Tree**, etc.

Statistics	First Edition	Second Edition	Third Edition
Number of Pages	10	24 (+140%)	35 (+46%)
Written Exercises	4	24 (+500%)	17+16* = 33 (+38%)
Programming Exercises	54	129 (+138%)	164 (+27%)

The breakdown of the number of programming exercises from each section is shown below:

Section	Title	Appearance	% in Chapter	% in Book
6.3	**Ad Hoc Strings Problems**	126	77%	8%
6.4	String Matching	13	8%	1%
6.5	**String Processing with DP**	17	10%	1%
6.6	Suffix Trie/Tree/Array	8	5%	≈ 1%

L-R: Dr Bill Poucher, Steven

Chapter 7

(Computational) Geometry

Let no man ignorant of geometry enter here.
— **Plato's Academy in Athens**

7.1 Overview and Motivation

(Computational[1]) Geometry is yet another topic that frequently appears in programming contests. Almost all ICPC problem sets have *at least one* geometry problem. If you are lucky, it will ask you for some geometry solution that you have learned before. Usually you draw the geometrical object(s) and then derive the solution from some basic geometric formulas. However, many geometry problems are the *computational* ones that require some complex algorithm(s).

In IOI, the existence of geometry-specific problems depends on the tasks chosen by the Scientific Committee that year. In recent years (2009-2012), IOI tasks do not feature *pure* geometry-specific problems. However, in the earlier years [67], every IOI contain one or two geometry related problems.

We have observed that geometry-related problems are usually not attempted during the early part of the contest time for *strategic reason* because the solutions for geometry-related problems have *lower* probability of getting Accepted (AC) during contest time compared to the solutions for other problem types in the problem set, e.g. Complete Search or Dynamic Programming problems. The typical issues with geometry problems are as follow:

- Many geometry problems have one and usually several tricky 'corner test cases', e.g. What if the lines are vertical (infinite gradient)?, What if the points are collinear?, What if the polygon is concave?, What if the convex hull of a set of points is the set of points itself?, etc. Therefore, it is usually a very good idea to test your team's geometry solution with lots of corner test cases before you submit it for judging.

- There is a possibility of having floating point precision errors that cause even a 'correct' algorithm to get a Wrong Answer (WA) response.

- The solutions for geometry problems usually involve *tedious* coding.

These reasons cause many contestants to view that spending precious minutes attempting *other* problem types in the problem set more worthwhile than attempting a geometry problem that has lower probability of acceptance.

---

[1]We differentiate between *pure* geometry problems and the *computational* geometry ones. Pure geometry problems can normally be solved by hand (pen and paper method). Computational geometry problems typically require running an algorithm using computer to obtain the solution.

However, another not-so-good reason for the lack of attempts for geometry problems is because the contestants are not well prepared.

- The contestants forget some important basic formulas or are unable to derive the required (more complex) formulas from the basic ones.

- The contestants do not prepare well-written library functions before contest and their attempts to code such functions during stressful contest environment end up with one, but usually several[2], bug(s). In ICPC, the top teams usually fill a sizeable part of their hard copy material (which they can bring into the contest room) with lots of geometry formulas and library functions.

The main aim of this chapter is therefore to increase the number of attempts (and also AC solutions) for geometry-related problems in programming contests. Study this chapter for some ideas on tackling (computational) geometry problems in ICPCs and IOIs. There are only two sections in this chapter.

In Section 7.2, we present many (it is impossible to enumerate all) English geometric terminologies[3] and various basic formulas for 0D, 1D, 2D, and 3D **geometry objects** commonly found in programming contests. This section can be used as a quick reference when contestants are given geometry problems and are not sure of certain terminologies or forget some basic formulas.

In Section 7.3, we discuss several algorithms on 2D **polygons**. There are several nice pre-written library routines which can differentiate good from average teams (contestants) like the algorithms for deciding if a polygon is convex or concave, deciding if a point is inside or outside a polygon, cutting a polygon with a straight line, finding the convex hull of a set of points, etc.

The implementations of the formulas and computational geometry algorithms shown in this chapter use the following techniques to increase the probability of acceptance:

1. We highlight the special cases that can potentially arise and/or choose the implementation that reduces the number of such special cases.

2. We try to avoid floating point operations (i.e. division, square root, and any other operations that can produce numerical errors) and work with precise integers whenever possible (i.e. integer additions, subtractions, multiplications).

3. If we really need to work with floating point, we do floating point equality test this way: `fabs(a - b) < EPS` where `EPS` is a small number[4] like `1e-9` instead of testing if `a == b`. When we need to check if a floating point number $x \geq 0.0$, we use `x > -EPS` (similarly to test if $x \leq 0.0$, we use `x < EPS`).

---

[2]As a reference, the library code on points, lines, circles, triangles, and polygons shown in this chapter require several iterations of bug fixes to ensure that as many (usually subtle) bugs and special cases are handled properly.

[3]ACM ICPC and IOI contestants come from various nationalities and backgrounds. Therefore, we would like to get everyone familiarized with the English geometric terminologies.

[4]Unless otherwise stated, this `1e-9` is the default value of EPS(ilon) that we use in this chapter.

7.2 Basic Geometry Objects with Libraries

7.2.1 0D Objects: Points

1. **Point** is the basic building block of higher dimensional geometry objects. In 2D Euclidean[5] space, points are usually represented with a struct in C/C++ (or Class in Java) with two[6] members: The **x** and **y** coordinates w.r.t origin, i.e. coordinate (0, 0). If the problem description uses integer coordinates, use `int`s; otherwise, use `double`s. In order to be generic, we use the floating-point version of `struct point` in this book. A default and user-defined constructors can be used to (slightly) simplify coding later.

```
// struct point_i { int x, y; };    // basic raw form, minimalist mode
struct point_i { int x, y;     // whenever possible, work with point_i
  point_i() { x = y = 0; }                    // default constructor
  point_i(int _x, int _y) : x(_x), y(_y) {} };        // user-defined

struct point { double x, y;   // only used if more precision is needed
  point() { x = y = 0.0; }                    // default constructor
  point(double _x, double _y) : x(_x), y(_y) {} };    // user-defined
```

2. Sometimes we need to sort the points. We can easily do that by overloading the less than operator inside `struct point` and use sorting library.

```
struct point { double x, y;
  point() { x = y = 0.0; }
  point(double _x, double _y) : x(_x), y(_y) {}
  bool operator < (point other) const { // override less than operator
    if (fabs(x - other.x) > EPS)                  // useful for sorting
      return x < other.x;         // first criteria , by x-coordinate
    return y < other.y; } };       // second criteria, by y-coordinate

// in int main(), assuming we already have a populated vector<point> P
  sort(P.begin(), P.end());    // comparison operator is defined above
```

3. Sometimes we need to test if two points are equal. We can easily do that by overloading the equal operator inside `struct point`.

```
struct point { double x, y;
  point() { x = y = 0.0; }
  point(double _x, double _y) : x(_x), y(_y) {}
  // use EPS (1e-9) when testing equality of two floating points
  bool operator == (point other) const {
   return (fabs(x - other.x) < EPS && (fabs(y - other.y) < EPS)); } };

// in int main()
  point P1(0, 0), P2(0, 0), P3(0, 1);
  printf("%d\n", P1 == P2);                                  // true
  printf("%d\n", P1 == P3);                                  // false
```

---

[5]For simplicity, the 2D and 3D Euclidean spaces are the 2D and 3D world that we encounter in real life.
[6]Add one more member, z, if you are working in 3D Euclidean space.

4. We can measure Euclidean distance[7] between two points by using the function below.

```
double dist(point p1, point p2) {                    // Euclidean distance
                    // hypot(dx, dy) returns sqrt(dx * dx + dy * dy)
  return hypot(p1.x - p2.x, p1.y - p2.y); }          // return double
```

5. We can rotate a point by angle[8] θ counter clockwise around origin $(0, 0)$ by using a rotation matrix:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} \times \begin{bmatrix} x \\ y \end{bmatrix}$$
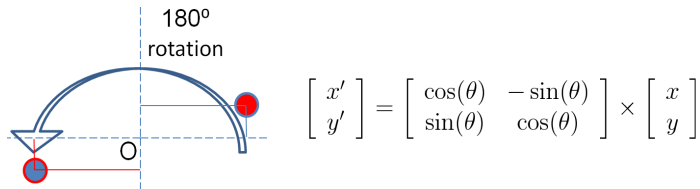
Figure 7.1: Rotating point $(10, 3)$ by 180 degrees counter clockwise around origin $(0, 0)$

```
// rotate p by theta degrees CCW w.r.t origin (0, 0)
point rotate(point p, double theta) {
  double rad = DEG_to_RAD(theta);    // multiply theta with PI / 180.0
  return point(p.x * cos(rad) - p.y * sin(rad),
               p.x * sin(rad) + p.y * cos(rad)); }
```

Exercise 7.2.1.1: Compute the Euclidean distance between point $(2, 2)$ and $(6, 5)$!

Exercise 7.2.1.2: Rotate a point $(10, 3)$ by 90 degrees counter clockwise around origin. What is the new coordinate of the rotated point? (easy to compute by hand).

Exercise 7.2.1.3: Rotate the same point $(10, 3)$ by 77 degrees counter clockwise around origin. What is the new coordinate of the rotated point? (this time you need to use calculator and the rotation matrix).

7.2.2   1D Objects: Lines

1. **Line** in 2D Euclidean space is the set of points whose coordinates satisfy a given linear equation $ax + by + c = 0$. Subsequent functions in this subsection assume that this linear equation has $b = 1$ for non vertical lines and $b = 0$ for vertical lines unless otherwise stated. Lines are usually represented with a struct in C/C++ (or Class in Java) with three members: The coefficients a, b, and c of that line equation.

```
struct line { double a, b, c; };          // a way to represent a line
```

2. We can compute the required line equation if we are given *at least* two points that pass through that line via the following function.

---

[7]The Euclidean distance between two points is simply the distance that can be measured with ruler. Algorithmically, it can be found with Pythagorean formula that we will see again in the subsection about triangle below. Here, we simply use a library function.

[8]Humans usually work with degrees, but many mathematical functions in most programming languages (e.g. C/C++/Java) work with radians. To convert an angle from degrees to radians, multiply the angle by $\frac{\pi}{180.0}$. To convert an angle from radians to degrees, multiply the angle with $\frac{180.0}{\pi}$.

```
// the answer is stored in the third parameter (pass by reference)
void pointsToLine(point p1, point p2, line &l) {
  if (fabs(p1.x - p2.x) < EPS) {              // vertical line is fine
    l.a = 1.0;   l.b = 0.0;   l.c = -p1.x;          // default values
  } else {
    l.a = -(double)(p1.y - p2.y) / (p1.x - p2.x);
    l.b = 1.0;                   // IMPORTANT: we fix the value of b to 1.0
    l.c = -(double)(l.a * p1.x) - p1.y;
} }
```

3. We can test whether two lines are *parallel* by checking if their coefficients a and b are the same. We can further test whether two lines are *the same* by checking if they are parallel and their coefficients c are the same (i.e. all three coefficients a, b, c are the same). Recall that in our implementation, we have fixed the value of coefficient b to 0.0 for all vertical lines and to 1.0 for all *non* vertical lines.

```
bool areParallel(line l1, line l2) {        // check coefficients a & b
  return (fabs(l1.a-l2.a) < EPS) && (fabs(l1.b-l2.b) < EPS); }

bool areSame(line l1, line l2) {            // also check coefficient c
  return areParallel(l1 ,l2) && (fabs(l1.c - l2.c) < EPS); }
```

4. If two lines[9] are not parallel (and therefore also not the same), they will intersect at a point. That intersection point (x, y) can be found by solving the system of two linear algebraic equations[10] with two unknowns: $a_1x + b_1y + c_1 = 0$ and $a_2x + b_2y + c_2 = 0$.

```
// returns true (+ intersection point) if two lines are intersect
bool areIntersect(line l1, line l2, point &p) {
  if (areParallel(l1, l2)) return false;          // no intersection
  // solve system of 2 linear algebraic equations with 2 unknowns
  p.x = (l2.b * l1.c - l1.b * l2.c) / (l2.a * l1.b - l1.a * l2.b);
  // special case: test for vertical line to avoid division by zero
  if (fabs(l1.b) > EPS) p.y = -(l1.a * p.x + l1.c);
  else                  p.y = -(l2.a * p.x + l2.c);
  return true; }
```

5. **Line Segment** is a line with two end points with *finite length*.

6. **Vector**[11] is a line segment (thus it has two end points and length/magnitude) with a *direction*. Usually[12], vectors are represented with a struct in C/C++ (or Class in Java) with two members: The **x** and **y** magnitude of the vector. The magnitude of the vector can be scaled if needed.

7. We can translate (move) a point w.r.t a vector as a vector describes the displacement magnitude in x and y-axis.

---

[9]To avoid confusion, please differentiate between line intersection versus line *segment* intersection.
[10]See Section 9.9 for the general solution for a system of linear equations.
[11]Do not confuse this with C++ STL `vector` or Java `Vector`.
[12]Another potential design strategy is to merge `struct point` with `struct vec` as they are similar.

```
struct vec { double x, y;   // name: 'vec' is different from STL vector
  vec(double _x, double _y) : x(_x), y(_y) {} };

vec toVec(point a, point b) {       // convert 2 points to vector a->b
  return vec(b.x - a.x, b.y - a.y); }

vec scale(vec v, double s) {        // nonnegative s = [<1 .. 1 .. >1]
  return vec(v.x * s, v.y * s); }                 // shorter.same.longer

point translate(point p, vec v) {       // translate p according to v
  return point(p.x + v.x , p.y + v.y); }
```

8. Given a point p and a line l (described by two points a and b), we can compute the minimum distance from p to l by first computing the location of point c in l that is closest to point p (see Figure 7.2—left) and then obtain the Euclidean distance between p and c. We can view point c as point a translated by a scaled magnitude u of vector ab, or $c = a + u \times ab$. To get u, we do scalar projection of vector ap onto vector ab by using dot product (see the dotted vector $ac = u \times ab$ in Figure 7.2—left). The short implementation of this solution is shown below.

```
double dot(vec a, vec b) { return (a.x * b.x + a.y * b.y); }

double norm_sq(vec v) { return v.x * v.x + v.y * v.y; }

// returns the distance from p to the line defined by
// two points a and b (a and b must be different)
// the closest point is stored in the 4th parameter (byref)
double distToLine(point p, point a, point b, point &c) {
  // formula: c = a + u * ab
  vec ap = toVec(a, p), ab = toVec(a, b);
  double u = dot(ap, ab) / norm_sq(ab);
  c = translate(a, scale(ab, u));               // translate a to c
  return dist(p, c); }          // Euclidean distance between p and c
```

Note that this is not the only way to get the required answer.
Solve **Exercise 7.2.2.10** for the alternative way.
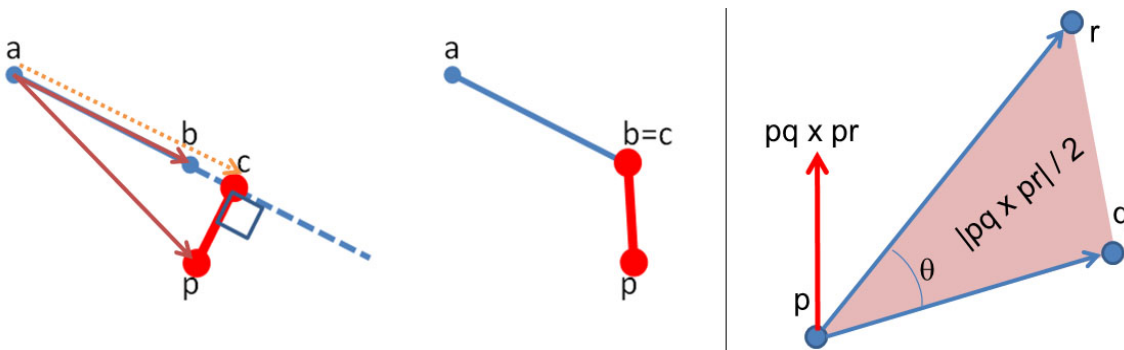
Figure 7.2: Distance to Line (left) and to Line Segment (middle); Cross Product (right)

9. If we are given a line *segment* instead (defined by two *end* points a and b), then the minimum distance from point p to line segment ab must also consider two special cases, the end points a and b of that line segment (see Figure 7.2—middle). The implementation is very similar to `distToLine` function above.

```
// returns the distance from p to the line segment ab defined by
// two points a and b (still OK if a == b)
// the closest point is stored in the 4th parameter (byref)
double distToLineSegment(point p, point a, point b, point &c) {
  vec ap = toVec(a, p), ab = toVec(a, b);
  double u = dot(ap, ab) / norm_sq(ab);
  if (u < 0.0) { c = point(a.x, a.y);                    // closer to a
    return dist(p, a); }         // Euclidean distance between p and a
  if (u > 1.0) { c = point(b.x, b.y);                    // closer to b
    return dist(p, b); }         // Euclidean distance between p and b
  return distToLine(p, a, b, c); }        // run distToLine as above
```

10. We can compute the angle *aob* given three points: a, o, and b, using dot product[13]. Since $oa \cdot ob = |oa| \times |ob| \times \cos(\theta)$, we have $theta = \arccos(oa \cdot ob/(|oa| \times |ob|))$.

```
double angle(point a, point o, point b) {  // returns angle aob in rad
  vec oa = toVector(o, a), ob = toVector(o, b);
  return acos(dot(oa, ob) / sqrt(norm_sq(oa) * norm_sq(ob))); }
```

11. Given a line defined by two points p and q, we can determine whether a point r is on the left/right side of the line, or whether the three points p, q, and r are collinear. This can be determined with *cross product*. Let pq and pr be the two vectors obtained from these three points. The cross product $pq \times pr$ result in another vector that is perpendicular to both pq and pr. The magnitude of this vector is equal to the area of the *parallelogram* that the vectors span[14]. If the magnitude is positive/zero/negative, then we know that $p \to q \to r$ is a left turn/collinear/right turn, respectively (see Figure 7.2—right). The left turn test is more famously known as the **CCW (Counter Clockwise) Test**.

```
double cross(vec a, vec b) { return a.x * b.y - a.y * b.x; }

// note: to accept collinear points, we have to change the '> 0'
// returns true if point r is on the left side of line pq
bool ccw(point p, point q, point r) {
  return cross(toVec(p, q), toVec(p, r)) > 0; }

// returns true if point r is on the same line as the line pq
bool collinear(point p, point q, point r) {
  return fabs(cross(toVec(p, q), toVec(p, r))) < EPS; }
```

Source code: `ch7_01_points_lines.cpp/java`

---

[13]`acos` is the C/C++ function name for mathematical function `arccos`.
[14]The area of triangle *pqr* is therefore *half* of the area of this parallelogram.

Exercise 7.2.2.1: A line can also be described with this mathematical equation: $y = mx + c$ where m is the 'gradient'/'slope' of the line and c is the 'y-intercept' constant.
Which form is better ($ax + by + c = 0$ or the slope-intercept form $y = mx + c$)? Why?

Exercise 7.2.2.2: Compute line equation that pass through two points (2, 2) and (4, 3)!

Exercise 7.2.2.3: Compute line equation that pass through two points (2, 2) and (2, 4)!

Exercise 7.2.2.4: Suppose we insist to use the other line equation: $y = mx + c$. Show how to compute the required line equation given two points that pass through that line! Try on two points (2, 2) and (2, 4) as in **Exercise 7.2.2.3**. Do you encounter any problem?

Exercise 7.2.2.5: We can also compute the line equation if we are given *one* point and the gradient/slope of that line. Show how to compute line equation given a point and gradient!

Exercise 7.2.2.6: Translate a point c (3, 2) according to a vector ab defined by two points: a (2, 2) and b (4, 3). What is the new coordinate of the point?

Exercise 7.2.2.7: Same as **Exercise 7.2.2.6** above, but now the magnitude of vector ab is reduced by *half*. What is the new coordinate of the point?

Exercise 7.2.2.8: Same as **Exercise 7.2.2.6** above, then rotate the resulting point by 90 degrees counter clockwise around origin. What is the new coordinate of the point?

Exercise 7.2.2.9: Rotate a point c (3, 2) by 90 degrees counter clockwise around origin, then translate the resulting point according to a vector ab. Vector ab is the same as in **Exercise 7.2.2.6** above. What is the new coordinate of the point? Is the result similar with the previous **Exercise 7.2.2.8** above? What can we learn from this phenomenon?

Exercise 7.2.2.10: Rotate a point c (3, 2) by 90 degrees counter clockwise but around point p (2, 1) (note that point p is *not* the origin). Hint: You need to translate the point.

Exercise 7.2.2.11: We can compute the location of point c in line l that is closest to point p by finding the other line l' that is perpendicular with line l and pass through point p. The closest point c is the intersection point between line l and l'. Now, how to obtain a line perpendicular to l? Are there special cases that we have to be careful with?

Exercise 7.2.2.12: Given a point p and a line l (described by two points a and b), show how to compute the location of a reflection point r of point p when mirrored against line l.

Exercise 7.2.2.13: Given three points: a (2, 2), o (2, 4), and b (4, 3), compute the angle aob in degrees!

Exercise 7.2.2.14: Determine if point r (35, 30) is on the left side of, collinear with, or is on the right side of a line that passes through two points p (3, 7) and q (11, 13).

7.2.3   2D Objects: Circles

1. **Circle** centered at coordinate (a, b) in a 2D Euclidean space with **radius** r is the set of all points (x, y) such that $(x - a)^2 + (y - b)^2 = r^2$.

2. To check if a point is inside, outside, or exactly on the border of a circle, we can use the following function. Modify this function a bit for the floating point version.

```
int insideCircle(point_i p, point_i c, int r) { // all integer version
    int dx = p.x - c.x, dy = p.y - c.y;
    int Euc = dx * dx + dy * dy, rSq = r * r;                    // all integer
    return Euc < rSq ? 0 : Euc == rSq ? 1 : 2; } //inside/border/outside
```
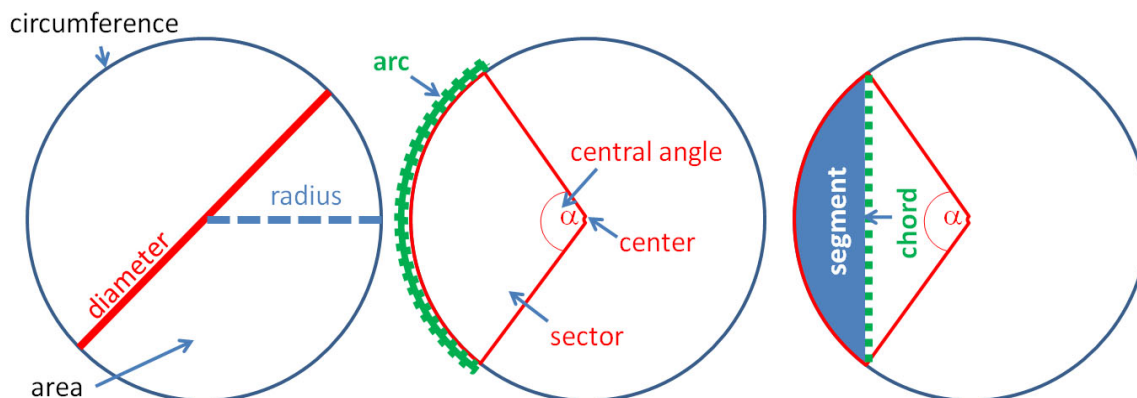

Figure 7.3: Circles

3. The constant **Pi** (π) is the ratio of *any* circle's circumference to its diameter. To avoid precision error, the safest value for programming contest if this constant π is not defined in the problem description is `pi = acos(-1.0)` or `pi = 2 * acos(0.0)`.

4. A circle with radius r has **diameter** $d = 2 \times r$ and **circumference** (or **perimeter**) $c = 2 \times \pi \times r$.

5. A circle with radius r has **area** $A = \pi \times r^2$

6. **Arc** of a circle is defined as a connected section of the circumference c of the circle. Given the central angle α (angle with vertex at the circle's center, see Figure 7.3—middle) in degrees, we can compute the length of the corresponding arc as $\frac{\alpha}{360.0} \times c$.

7. **Chord** of a circle is defined as a line segment whose endpoints lie on the circle[15]. A circle with radius r and a central angle α in degrees (see Figure 7.3—right) has the corresponding chord with length $sqrt(2 \times r^2 \times (1 - cos(\alpha)))$. This can be derived from the **Law of Cosines**—see the explanation of this law in the discussion about Triangles later. Another way to compute the length of chord given r and α is to use Trigonometry: $2 \times r \times sin(\alpha/2)$. Trigonometry is also discussed below.

8. **Sector** of a circle is defined as a region of the circle enclosed by two radius and an arc lying between the two radius. A circle with area A and a central angle α (in degrees)—see Figure 7.3, middle—has the corresponding sector area $\frac{\alpha}{360.0} \times A$.

9. **Segment** of a circle is defined as a region of the circle enclosed by a chord and an arc lying between the chord's endpoints (see Figure 7.3—right). The area of a segment can be found by subtracting the area of the corresponding sector from the area of an isosceles triangle with sides: r, r, and chord-length.

---

[15]Diameter is the longest chord in a circle.

10. Given 2 points on the circle ($p1$ and $p2$) and radius r of the corresponding circle, we can determine the location of the centers ($c1$ and $c2$) of the two possible circles (see Figure 7.4). The code is shown in **Exercise 7.2.3.1** below.
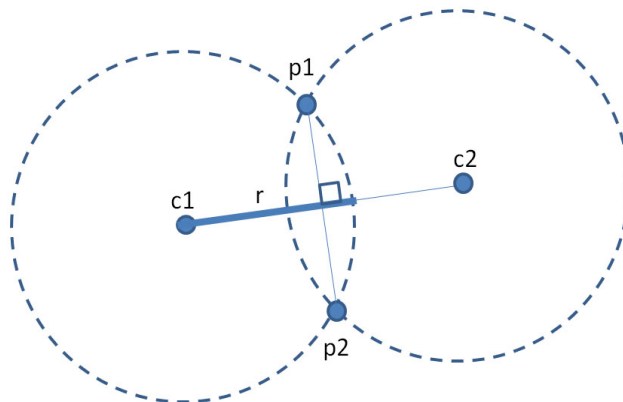
Figure 7.4: Circle Through 2 Points and Radius

Source code: `ch7_02_circles.cpp/java`

Exercise 7.2.3.1: Explain what is computed by the code below!

```
bool circle2PtsRad(point p1, point p2, double r, point &c) {
  double d2 = (p1.x - p2.x) * (p1.x - p2.x) +
              (p1.y - p2.y) * (p1.y - p2.y);
  double det = r * r / d2 - 0.25;
  if (det < 0.0) return false;
  double h = sqrt(det);
  c.x = (p1.x + p2.x) * 0.5 + (p1.y - p2.y) * h;
  c.y = (p1.y + p2.y) * 0.5 + (p2.x - p1.x) * h;
  return true; }          // to get the other center, reverse p1 and p2
```

7.2.4   2D Objects: Triangles

1. **Triangle** (three angles) is a polygon with three vertices and three edges.
   There are several types of triangles:
   a. **Equilateral**: Three equal-length edges and all inside (interior) angles are 60 degrees;
   b. **Isosceles**: Two edges have the same length and two interior angles are the same.
   c. **Scalene**: All edges have different length;
   d. **Right**: *One* of its interior angle is 90 degrees (or a **right angle**).

2. A triangle with base b and height h has **area** $A = 0.5 \times b \times h$.

3. A triangle with three sides: a, b, c has **perimeter** $p = a + b + c$ and **semi-perimeter** $s = 0.5 \times p$.

4. A triangle with 3 sides: a, b, c and semi-perimeter s has area $A = sqrt(s \times (s - a) \times (s - b) \times (s - c))$. This formula is called the **Heron's Formula**.
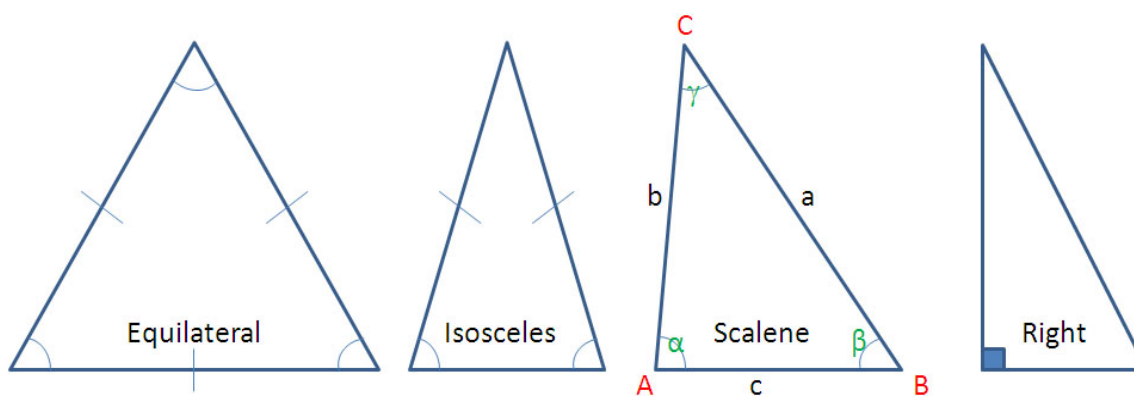
Figure 7.5: Triangles

5. A triangle with area A and semi-perimeter s has an **inscribed circle (incircle)** with radius $r = A/s$.

```
double rInCircle(double ab, double bc, double ca) {
  return area(ab, bc, ca) / (0.5 * perimeter(ab, bc, ca)); }

double rInCircle(point a, point b, point c) {
  return rInCircle(dist(a, b), dist(b, c), dist(c, a)); }
```

6. The center of incircle is the meeting point between the triangle's *angle bisectors* (see Figure 7.6—left). We can get the center if we have two angle bisectors and find their intersection point. The implementation is shown below:

```
// assumption: the required points/lines functions have been written
// returns 1 if there is an inCircle center, returns 0 otherwise
// if this function returns 1, ctr will be the inCircle center
// and r is the same as rInCircle
int inCircle(point p1, point p2, point p3, point &ctr, double &r) {
  r = rInCircle(p1, p2, p3);
  if (fabs(r) < EPS) return 0;                      // no inCircle center

  line l1, l2;                       // compute these two angle bisectors
  double ratio = dist(p1, p2) / dist(p1, p3);
  point p = translate(p2, scale(toVec(p2, p3), ratio / (1 + ratio)));
  pointsToLine(p1, p, l1);

  ratio = dist(p2, p1) / dist(p2, p3);
  p = translate(p1, scale(toVec(p1, p3), ratio / (1 + ratio)));
  pointsToLine(p2, p, l2);

  areIntersect(l1, l2, ctr);               // get their intersection point
  return 1; }
```

7. A triangle with 3 sides: a, b, c and area A has an **circumscribed circle (circumcircle)** with radius $R = a \times b \times c/(4 \times A)$.
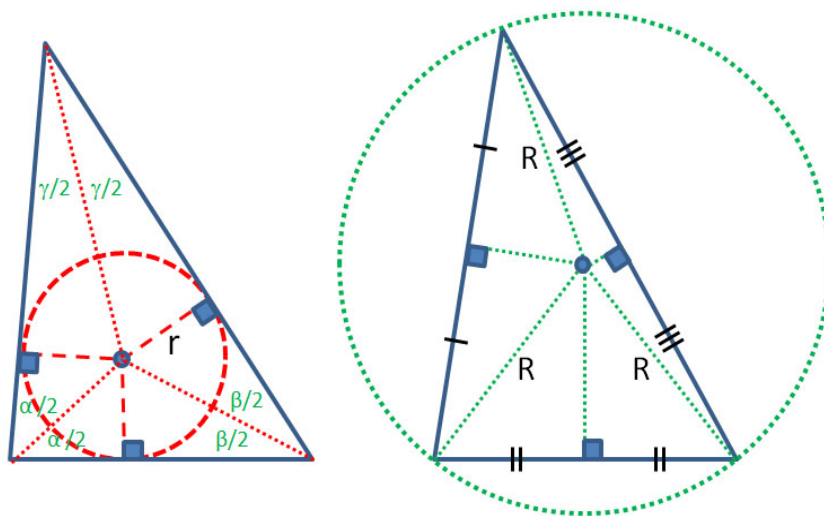
Figure 7.6: Incircle and Circumcircle of a Triangle

```
double rCircumCircle(double ab, double bc, double ca) {
  return ab * bc * ca / (4.0 * area(ab, bc, ca)); }

double rCircumCircle(point a, point b, point c) {
  return rCircumCircle(dist(a, b), dist(b, c), dist(c, a)); }
```

8. The center of circumcircle is the meeting point between the triangle's *perpendicular bisectors* (see Figure 7.6—right).

9. To check if three line segments of length a, b and c can form a triangle, we can simply check these *triangle inequalities*: $(a + b > c)$ && $(a + c > b)$ && $(b + c > a)$.
   If the result is false, then the three line segments cannot form a triangle.
   If the three lengths are sorted, with a being the smallest and c the largest, then we can simplify the check to just $(a + b > c)$.

10. When we study triangle, we should not forget **Trigonometry**—a study about the relationships between triangle sides and the angles between sides.

    In Trigonometry, the **Law of Cosines** (a.k.a. the **Cosine Formula** or the **Cosine Rule**) is a statement about a general triangle that relates the lengths of its sides to the cosine of one of its angles. See the scalene (middle) triangle in Figure 7.5. With the notations described there, we have: $c^2 = a^2 + b^2 - 2 \times a \times b \times cos(\gamma)$, or $\gamma = acos(\frac{a^2 + b^2 - c^2}{2 \times a \times b})$. The formula for the other two angles α and β are similarly defined.

11. In Trigonometry, the **Law of Sines** (a.k.a. the **Sine Formula** or the **Sine Rule**) is an equation relating the lengths of the sides of an arbitrary triangle to the sines of its angle. See the scalene (middle) triangle in Figure 7.5. With the notations described there and R is the radius of its circumcircle, we have: $\frac{a}{sin(\alpha)} = \frac{b}{sin(\beta)} = \frac{c}{sin(\gamma)} = 2R$.

12. The **Pythagorean Theorem** specializes the Law of Cosines. This theorem only applies to right triangles. If the angle γ is a right angle (of measure 90° or $\pi/2$ radians), then $cos(\gamma) = 0$, and thus the Law of Cosines reduces to: $c^2 = a^2 + b^2$. Pythagorean theorem is used in finding the Euclidean distance between two points shown earlier.

13. The **Pythagorean Triple** is a triple with three positive integers a, b, and c—commonly written as (a, b, c)—such that $a^2 + b^2 = c^2$. A well-known example is $(3, 4, 5)$. If (a, b, c) is a Pythagorean triple, then so is (ka, kb, kc) for any positive integer k. A Pythagorean Triple describes the integer lengths of the three sides of a Right Triangle.

Source code: `ch7_03_triangles.cpp/java`

Exercise 7.2.4.1: Let a, b, and c of a triangle be 2^{18}, 2^{18}, and 2^{18}. Can we compute the area of this triangle with Heron's formula as shown in point 4 above without experiencing overflow (assuming that we use 64-bit integers)? What should we do to avoid this issue?

Exercise 7.2.4.2*: Implement the code to find the center of the circumCircle of three points a, b, and c. The function structure is similar as function `inCircle` shown in this section.

Exercise 7.2.4.3*: Implement another code to check if a point d is inside the circumCircle of three points a, b, and c.

7.2.5   2D Objects: Quadrilaterals
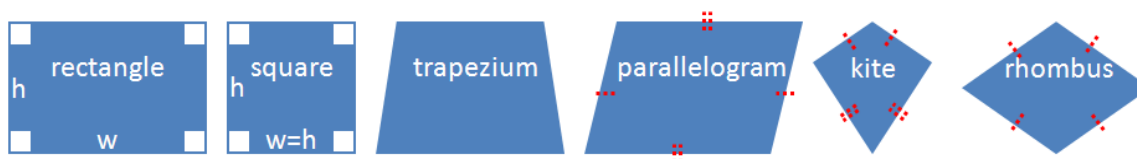
Figure 7.7: Quadrilaterals

1. **Quadrilateral** or **Quadrangle** is a polygon with four edges (and four vertices). The term 'polygon' itself is described in more details below (Section 7.3). Figure 7.7 shows a few examples of Quadrilateral objects.

2. **Rectangle** is a polygon with four edges, four vertices, and four right angles.

3. A rectangle with width w and height h has **area** $A = w \times h$ and **perimeter** $p = 2 \times (w + h)$.

4. **Square** is a special case of a rectangle where $w = h$.

5. **Trapezium** is a polygon with four edges, four vertices, and one pair of parallel edges. If the two non-parallel sides have the same length, we have an **Isosceles Trapezium**.

6. A trapezium with a pair of parallel edges of lengths $w1$ and $w2$; and a height h between both parallel edges has area $A = 0.5 \times (w1 + w2) \times h$.

7. **Parallelogram** is a polygon with four edges and four vertices. Moreover, the opposite sides must be parallel.

8. **Kite** is a quadrilateral which has two pairs of sides of the same length which are adjacent to each other. The area of a kite is $diagonal_1 \times diagonal_2/2$.

9. **Rhombus** is a special parallelogram where every side has equal length. It is also a special case of kite where every side has equal length.

Remarks about 3D Objects

Programming contest problems involving 3D objects are rare. But when such a problem does appear in a problem set, it can be one of the hardest. In the list of programming exercises below, we include an initial list of problems involving 3D objects.

Programming Exercises related to Basic Geometry:

- Points and Lines:
    1. UVa 00152 - Tree's a Crowd (sort the 3D points first)
    2. UVa 00191 - Intersection (line segment intersection)
    3. UVa 00378 - Intersecting Lines (use `areParallel`, `areSame`, `areIntersect`)
    4. UVa 00587 - There's treasure everywhere (Euclidean distance `dist`)
    5. UVa 00833 - Water Falls (recursive check, use the `ccw` tests)
    6. UVa 00837 - Light and Transparencies (line segments, sort x-coords first)
    7. **UVa 00920 - Sunny Mountains *** (Euclidean distance `dist`)
    8. UVa 01249 - Euclid (LA 4601, Southeast USA Regional 2009, vector)
    9. UVa 10242 - Fourth Point (`toVector`; `translate` points w.r.t that vector)
    10. *UVa 10250 - The Other Two Trees* (vector, rotation)
    11. **UVa 10263 - Railway *** (use `distToLineSegment`)
    12. UVa 10357 - Playball (Euclidean distance `dist`, simple Physics simulation)
    13. UVa 10466 - How Far? (Euclidean distance `dist`)
    14. UVa 10585 - Center of symmetry (sort the points)
    15. *UVa 10832 - Yoyodyne Propulsion ...* (3D Euclidean distance; simulation)
    16. UVa 10865 - Brownie Points (points and quadrants, simple)
    17. UVa 10902 - Pick-up sticks (line segment intersection)
    18. **UVa 10927 - Bright Lights *** (sort points by gradient, Euclidean distance)
    19. UVa 11068 - An Easy Task (simple 2 linear equations with 2 unknowns)
    20. UVa 11343 - Isolated Segments (line segment intersection)
    21. UVa 11505 - Logo (Euclidean distance `dist`)
    22. *UVa 11519 - Logo 2* (vectors and angles)
    23. *UVa 11894 - Genius MJ* (about rotating and translating points)
- Circles (only):
    1. *UVa 01388 - Graveyard* (divide the circle into n sectors first and then into $(n + m)$ sectors)
    2. **UVa 10005 - Packing polygons *** (complete search; use `circle2PtsRad` discussed in Chapter 7)
    3. UVa 10136 - Chocolate Chip Cookies (similar to UVa 10005)
    4. UVa 10180 - Rope Crisis in Ropeland (closest point from AB to origin; arc)
    5. UVa 10209 - Is This Integration? (square, arcs, similar to UVa 10589)
    6. UVa 10221 - Satellites (finding arc and chord length of a circle)
    7. *UVa 10283 - The Kissing Circles* (derive the formula)
    8. UVa 10432 - Polygon Inside A Circle (area of n-sided reg-polygon in circle)
    9. UVa 10451 - Ancient ... (inner/outer circle of n-sided reg polygon)
    10. UVa 10573 - Geometry Paradox (there is no 'impossible' case)
    11. **UVa 10589 - Area *** (check if point is inside intersection of 4 circles)

12. **UVa 10678 - The Grazing Cows *** (area of an *ellipse*, generalization of the formula for area of a circle)
13. *UVa 12578 - 10:6:2* (area of rectangle and circle)

- Triangles (plus Circles)

    1. UVa 00121 - Pipe Fitters (use Pythagorean theorem; grid)
    2. UVa 00143 - Orchard Trees (count integer points in triangle; precision issue)
    3. UVa 00190 - Circle Through Three ... (triangle's circumcircle)
    4. UVa 00375 - Inscribed Circles and ... (triangle's incircle**s**!)
    5. UVa 00438 - The Circumference of ... (triangle's circumcircle)
    6. UVa 10195 - The Knights Of The ... (triangle's incircle, Heron's formula)
    7. UVa 10210 - Romeo & Juliet (basic trigonometry)
    8. UVa 10286 - The Trouble with a ... (Law of Sines)
    9. UVa 10347 - Medians (given 3 medians of a triangle, find its area)
    10. UVa 10387 - Billiard (expanding surface, *trigonometry*)
    11. UVa 10522 - Height to Area (derive the formula, uses Heron's formula)
    12. **UVa 10577 - Bounding box *** (get center+radius of outer circle from 3 points, get all vertices, get the min-x/max-x/min-y/max-y of the polygon)
    13. *UVa 10792 - The Laurel-Hardy Story* (derive the trigonometry formulas)
    14. UVa 10991 - Region (Heron's formula, Law of Cosines, area of sector)
    15. **UVa 11152 - Colourful ... *** (triangle's (in/circum)circle; Heron's formula)
    16. *UVa 11164 - Kingdom Division* (use Triangle properties)
    17. *UVa 11281 - Triangular Pegs in ...* (the min bounding circle of a non obtuse triangle is its circumcircle; if the triangle is obtuse, the the radii of the min bounding circle is the largest side of the triangle)
    18. *UVa 11326 - Laser Pointer* (trigonometry, tangent, reflection trick)
    19. *UVa 11437 - Triangle Fun* (hint: $\frac{1}{7}$)
    20. UVa 11479 - Is this the easiest problem? (property check)
    21. UVa 11579 - Triangle Trouble (sort; greedily check if three successive sides satisfy triangle inequality and if it is the largest triangle found so far)
    22. UVa 11854 - Egypt (Pythagorean theorem/triple)
    23. **UVa 11909 - Soya Milk *** (Law of Sines (or tangent); two possible cases!)
    24. UVa 11936 - The Lazy Lumberjacks (see if 3 sides form a valid triangle)

- Quadrilaterals

    1. UVa 00155 - All Squares (recursive counting)
    2. **UVa 00460 - Overlapping Rectangles *** (rectangle-rectangle intersection)
    3. UVa 00476 - Points in Figures: ... (similar to UVa 477 and 478)
    4. UVa 00477 - Points in Figures: ... (similar to UVa 476 and 478)
    5. **UVa 11207 - The Easiest Way *** (cutting rectangle into 4-equal-sized squares)
    6. UVa 11345 - Rectangles (rectangle-rectangle intersection)
    7. UVa 11455 - Behold My Quadrangle (property check)
    8. UVa 11639 - Guard the Land (rectangle-rectangle intersection, use flag array)
    9. *UVa 11800 - Determine the Shape* (use `next_permutation` to help you try all possible 4! = 24 permutations of 4 points; check if they can satisfy square, rectangle, rhombus, parallelogram, trapezium, in that order)
    10. **UVa 11834 - Elevator *** (packing two circles in a rectangle)
    11. *UVa 12256 - Making Quadrilaterals* (LA 5001, KualaLumpur 10, start with three sides of 1, 1, 1, then the fourth side onwards must be the sum of the previous three to make a line; repeat until we reach the n-th side)

- 3D Objects
    1. **UVa 00737 - Gleaming the Cubes *** (cube and cube intersection)
    2. **UVa 00815 - Flooded *** (volume, greedy, sort by height, simulation)
    3. **UVa 10297 - Beavergnaw *** (cones, cylinders, volumes)

Profile of Algorithm Inventor

Pythagoras of Samos ($\approx$ 500 BC) was a Greek mathematician and philosopher born on the island of Samos. He is best known for the Pythagorean theorem involving right triangle.

Euclid of Alexandria ($\approx$ 300 BC) was a Greek mathematician, the 'Father of Geometry'. He was from the city of Alexandria. His most influential work in mathematics (especially geometry) is the 'Elements'. In the 'Elements', Euclid deduced the principles of what is now called Euclidean geometry from a small set of axioms.

Heron of Alexandria ($\approx$ 10-70 AD) was an ancient Greek mathematician from the city of Alexandria, Roman Egypt—the same city as Euclid. His name is closely associated with his formula for finding the area of a triangle from its side lengths.

Ronald Lewis Graham (born 1935) is an American mathematician. In 1972, he invented the Graham's scan algorithm for finding convex hull of a finite set of points in the plane. There are now many other algorithm variants and improvements for finding convex hull.

7.3   Algorithm on Polygon with Libraries

Polygon is a plane figure that is bounded by a closed path (path that starts and ends at the same vertex) composed of a finite sequence of straight line segments. These segments are called edges or sides. The point where two edges meet is the polygon's vertex or corner. Polygon is a source of many (computational) geometry problems as it allows the problem author to present more realistic objects than the ones discussed in Section 7.2.

7.3.1   Polygon Representation

The standard way to represent a polygon is to simply enumerate the vertices of the polygon in either clockwise or counter clockwise order, with the first vertex being equal to the last vertex (some of the functions mentioned later in this section require this arrangement, see **Exercise 7.3.4.1***). In this book, our default vertex ordering is counter clockwise. The resulting polygon after executing the code below is shown in Figure 7.8—right.

```
// 6 points, entered in counter clockwise order, 0-based indexing
vector<point> P;
P.push_back(point(1, 1));  // P0
P.push_back(point(3, 3));  // P1
P.push_back(point(9, 1));  // P2
P.push_back(point(12, 4)); // P3
P.push_back(point(9, 7));  // P4
P.push_back(point(1, 7));  // P5
P.push_back(P[0]); // important: loop back
```

7.3.2   Perimeter of a Polygon

The perimeter of a polygon (either convex or concave) with n vertices given in some order (either clockwise or counter-clockwise) can be computed via this simple function below.

```
// returns the perimeter, which is the sum of Euclidian distances
// of consecutive line segments (polygon edges)
double perimeter(const vector<point> &P) {
  double result = 0.0;
  for (int i = 0; i < (int)P.size()-1; i++)  // remember that P[0] = P[n-1]
    result += dist(P[i], P[i+1]);
  return result; }
```

7.3.3   Area of a Polygon

The signed area A of (either convex or concave) polygon with n vertices given in some order (either clockwise or counter-clockwise) can be found by computing the determinant of the matrix as shown below. This formula can be easily written into the library code.

$$A = \frac{1}{2} \times \begin{vmatrix} x_0 & y_0 \\ x_1 & y_1 \\ x_2 & y_2 \\ \cdots & \cdots \\ x_{n-1} & y_{n-1} \end{vmatrix} = \frac{1}{2} \times (x_0 \times y_1 + x_1 \times y_2 + \ldots + x_{n-1} \times y_0 - x_1 \times y_0 - x_2 \times y_1 - \ldots - x_0 \times y_{n-1})$$

```
// returns the area, which is half the determinant
double area(const vector<point> &P) {
  double result = 0.0, x1, y1, x2, y2;
  for (int i = 0; i < (int)P.size()-1; i++) {
    x1 = P[i].x; x2 = P[i+1].x;
    y1 = P[i].y; y2 = P[i+1].y;
    result += (x1 * y2 - x2 * y1);
  }
  return fabs(result) / 2.0; }
```

7.3.4 Checking if a Polygon is Convex

A polygon is said to be **Convex** if any line segment drawn inside the polygon does not intersect any edge of the polygon. Otherwise, the polygon is called **Concave**.
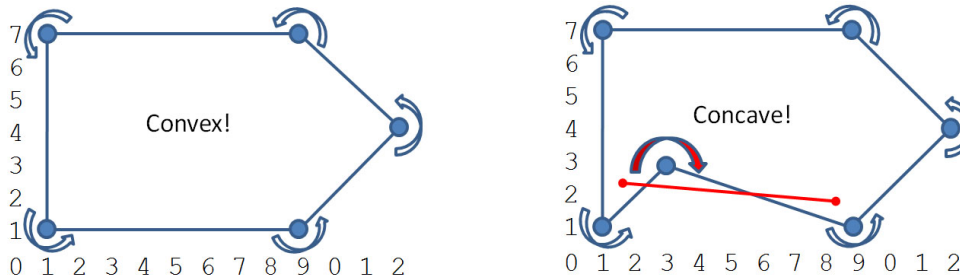
Figure 7.8: Left: Convex Polygon, Right: Concave Polygon

However, to test if a polygon is convex, there is an easier computational approach than "trying to check if all line segments can be drawn inside the polygon". We can simply check whether all three consecutive vertices of the polygon form the same turns (all left turns/ccw if the vertices are listed in counter clockwise order or all right turn/cw if the vertices are listed in clockwise order). If we can find at least one triple where this is false, then the polygon is concave (see Figure 7.8).

```
bool isConvex(const vector<point> &P) {          // returns true if all three
  int sz = (int)P.size();    // consecutive vertices of P form the same turns
  if (sz <= 3) return false;    // a point/sz=2 or a line/sz=3 is not convex
  bool isLeft = ccw(P[0], P[1], P[2]);                // remember one result
  for (int i = 1; i < sz-1; i++)              // then compare with the others
    if (ccw(P[i], P[i+1], P[(i+2) == sz ? 1 : i+2]) != isLeft)
      return false;               // different sign -> this polygon is concave
  return true; }                            // this polygon is convex
```

Exercise 7.3.4.1*: Which part of the code above that you should modify to accept collinear points? Example: Polygon {(0,0), (2,0), (4,0), (2,2), (0,0)} should be treated as convex.

Exercise 7.3.4.2*: If the first vertex is not repeated as the last vertex, will the function `perimeter`, `area`, and `isConvex` presented as above work correctly?

7.3.5 Checking if a Point is Inside a Polygon

Another common test performed on a polygon P is to check if a point pt is inside or outside polygon P. The following function that implements 'winding number algorithm' allows such check for *either* convex or concave polygon. It works by computing the sum of angles between three points: $\{P[i], pt, P[i+1]\}$ where $(P[i]$-$P[i+1])$ are consecutive sides of polygon P, taking care of left turns (add the angle) and right turns (subtract the angle) respectively. If the final sum is 2π (360 degrees), then pt is inside polygon P (see Figure 7.9).
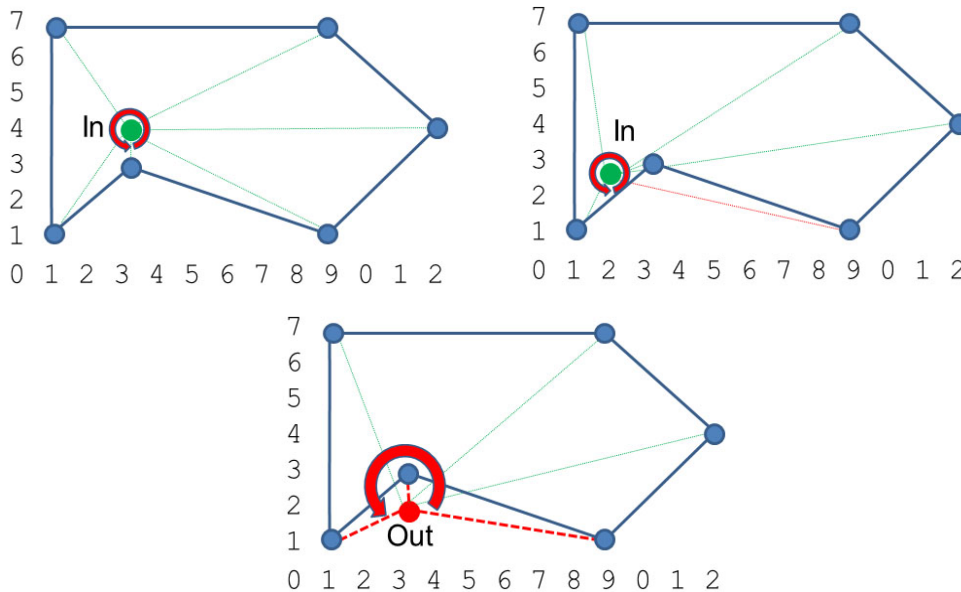
Figure 7.9: Top Left: inside, Top Right: also inside, Bottom: outside

```
// returns true if point p is in either convex/concave polygon P
bool inPolygon(point pt, const vector<point> &P) {
  if ((int)P.size() == 0) return false;
  double sum = 0;     // assume the first vertex is equal to the last vertex
  for (int i = 0; i < (int)P.size()-1; i++) {
    if (ccw(pt, P[i], P[i+1]))
        sum += angle(P[i], pt, P[i+1]);                    // left turn/ccw
    else sum -= angle(P[i], pt, P[i+1]); }                 // right turn/cw
  return fabs(fabs(sum) - 2*PI) < EPS; }
```

Exercise 7.9.1*: What happen to the `inPolygon` routine if point `pt` is on one of the edge of polygon P, e.g. `pt = P[0]` or `pt` is the mid-point between `P[0]` and `P[1]`, etc? What should be done to address that situation?

Exercise 7.9.2*: Discuss the pros and the cons of the following alternative methods for testing if a point is inside a polygon:

1. Triangulate a convex polygon into triangles and check if the sum of triangle areas equal to the area of the convex polygon.

2. Ray casting algorithm: We draw a ray from the point to any fixed direction so that the ray intersects the edge(s) of the polygon. If there are odd/even number of intersections, the point is inside/outside, respectively.

7.3.6 Cutting Polygon with a Straight Line

Another interesting thing that we can do with a *convex* polygon (see **Exercise 7.3.6.2*** for concave polygon) is to cut it into two convex sub-polygons with a straight line defined with two points a and b. See some programming exercises listed below that use this function.

The basic idea of the following `cutPolygon` routine is to iterate through the vertices of the original polygon Q one by one. If line ab and polygon vertex v form a left turn (which implies that v is on the left side of the line ab), we put v inside the new polygon P. Once we find a polygon edge that intersects with the line ab, we use that intersection point as part of the new polygon P (see Figure 7.10—left, point 'C'). We then skip the next few vertices of Q that are located on the right side of line ab. Sooner or later, we will revisit another polygon edge that intersect with line ab again (see Figure 7.10—left, point 'D' which happens to be one of the original vertex of polygon Q). We continue appending vertices of Q into P again because we are now on the left side of line ab again. We stop when we have returned to the starting vertex and returns the resulting polygon P (see Figure 7.10—right).
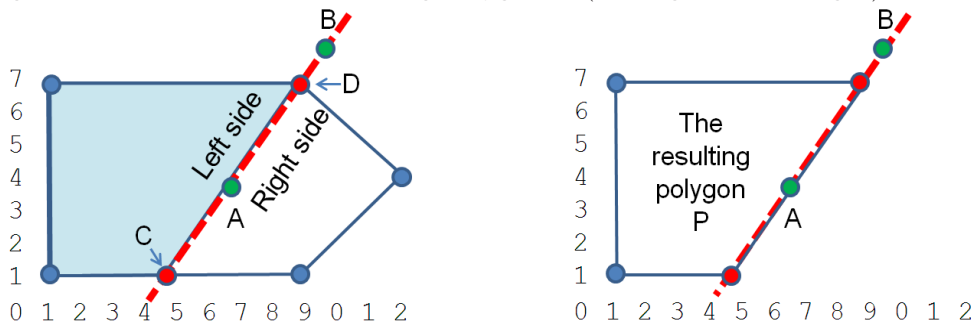
Figure 7.10: Left: Before Cut, Right: After Cut

```
// line segment p-q intersect with line A-B.
point lineIntersectSeg(point p, point q, point A, point B) {
  double a = B.y - A.y;
  double b = A.x - B.x;
  double c = B.x * A.y - A.x * B.y;
  double u = fabs(a * p.x + b * p.y + c);
  double v = fabs(a * q.x + b * q.y + c);
  return point((p.x * v + q.x * u) / (u+v), (p.y * v + q.y * u) / (u+v)); }

// cuts polygon Q along the line formed by point a -> point b
// (note: the last point must be the same as the first point)
vector<point> cutPolygon(point a, point b, const vector<point> &Q) {
  vector<point> P;
  for (int i = 0; i < (int)Q.size(); i++) {
    double left1 = cross(toVec(a, b), toVec(a, Q[i])), left2 = 0;
    if (i != (int)Q.size()-1) left2 = cross(toVec(a, b), toVec(a, Q[i+1]));
    if (left1 > -EPS) P.push_back(Q[i]);       // Q[i] is on the left of ab
    if (left1 * left2 < -EPS)          // edge (Q[i], Q[i+1]) crosses line ab
      P.push_back(lineIntersectSeg(Q[i], Q[i+1], a, b));
  }
  if (!P.empty() && !(P.back() == P.front()))
    P.push_back(P.front());          // make P's first point = P's last point
  return P; }
```

To further help readers to understand these algorithms on polygon, we have build a visualization tool for the third edition of this book. The reader can draw their own polygon and asks the tool to visually explain the algorithm on polygon discussed in this section.

Visualization: `www.comp.nus.edu.sg/~stevenha/visualization/polygon.html`

Exercise 7.3.6.1: This `cutPolygon` function only returns the left side of the polygon Q after cutting it with line ab. What should we do if we want the right side instead?

Exercise 7.3.6.2*: What happen if we run `cutPolygon` function on a *concave* polygon?

7.3.7 Finding the Convex Hull of a Set of Points

The **Convex Hull** of a set of points P is the smallest convex polygon $CH(P)$ for which each point in P is either on the boundary of $CH(P)$ or in its interior. Imagine that the points are nails on a flat 2D plane and we have a long enough rubber band that can enclose all the nails. If this rubber band is released, it will try to enclose as small an area as possible. That area is the area of the convex hull of these set of points/nails (see Figure 7.11). Finding convex hull of a set of points has natural applications in *packing* problems.
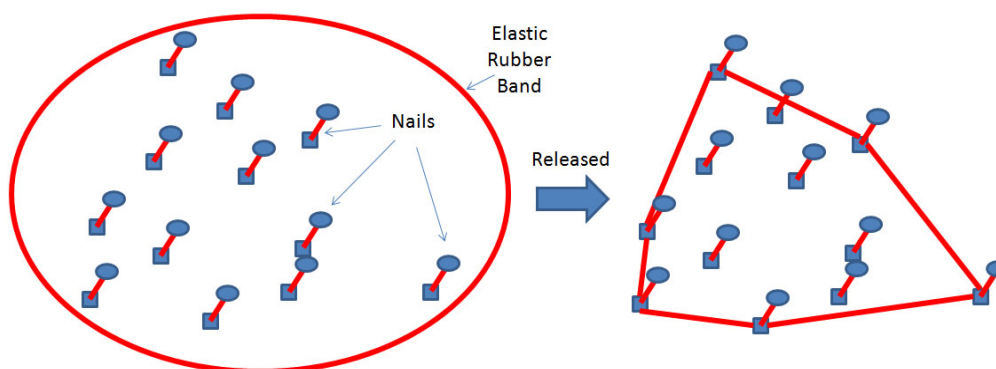
Figure 7.11: Rubber Band Analogy for Finding Convex Hull

As every vertex in $CH(P)$ is a vertex in the set of points P, the algorithm for finding convex hull is essentially an algorithm to decide which points in P should be chosen as part of the convex hull. There are several convex hull finding algorithms available. In this section, we choose the $O(n \log n)$ Ronald *Graham's Scan* algorithm.

Graham's scan algorithm first sorts all the n points of P where the first point does not have to be replicated as the last point (see Figure 7.12.A) based on their angles w.r.t a point called pivot. In our example, we pick the bottommost and rightmost point in P as pivot. After sorting based on angles w.r.t this pivot, we can see that edge 0-1, 0-2, 0-3, ..., 0-10, and 0-11 are in counter clockwise order (see point 1 to 11 w.r.t point 0 in Figure 7.12.B)!

```
point pivot(0, 0);
bool angleCmp(point a, point b) {                    // angle-sorting function
  if (collinear(pivot, a, b))                                 // special case
    return dist(pivot, a) < dist(pivot, b);     // check which one is closer
  double d1x = a.x - pivot.x, d1y = a.y - pivot.y;
  double d2x = b.x - pivot.x, d2y = b.y - pivot.y;
  return (atan2(d1y, d1x) - atan2(d2y, d2x)) < 0; }   // compare two angles
```

```
vector<point> CH(vector<point> P) {   // the content of P may be reshuffled
  int i, j, n = (int)P.size();
  if (n <= 3) {
    if (!(P[0] == P[n-1])) P.push_back(P[0]); // safeguard from corner case
    return P; }                        // special case, the CH is P itself

  // first, find P0 = point with lowest Y and if tie: rightmost X
  int P0 = 0;
  for (i = 1; i < n; i++)
    if (P[i].y < P[P0].y || (P[i].y == P[P0].y && P[i].x > P[P0].x))
      P0 = i;

  point temp = P[0]; P[0] = P[P0]; P[P0] = temp;    // swap P[P0] with P[0]

  // second, sort points by angle w.r.t. pivot P0
  pivot = P[0];                        // use this global variable as reference
  sort(++P.begin(), P.end(), angleCmp);            // we do not sort P[0]
// to be continued
```
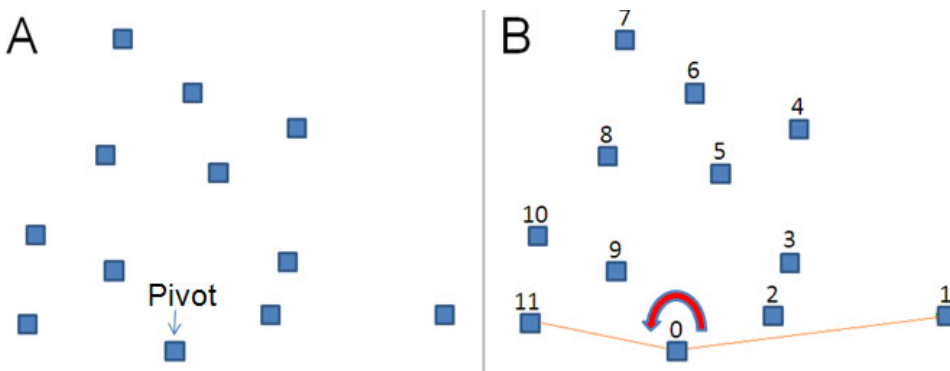

Figure 7.12: Sorting Set of 12 Points by Their Angles w.r.t a Pivot (Point 0)

Then, this algorithm maintains a stack S of candidate points. Each point of P is pushed *once* on to S and points that are not going to be part of $CH(P)$ will be eventually popped from S. Graham's Scan maintains this invariant: The top three items in stack S must always make a left turn (which is a basic property of a convex polygon).

Initially we insert these three points, point N-1, 0, and 1. In our example, the stack initially contains (bottom) 11-0-1 (top). This always form a left turn.

Now, examine Figure 7.13.C. Here, we try to insert point 2 and 0-1-2 is a left turn, so we accept point 2. Stack S is now (bottom) 11-0-1-2 (top).

Next, examine Figure 7.13.D. Here, we try to insert point 3 and 1-2-3 is a *right* turn. This means, if we accept the point before point 3, which is point 2, we will not have a convex polygon. So we have to pop point 2. Stack S is now (bottom) 11-0-1 (top) again. Then we re-try inserting point 3. Now 0-1-3, the *current* top three items in stack S form a left turn, so we accept point 3. Stack S is now (bottom) 11-0-1-3 (top).

We repeat this process until all vertices have been processed (see Figure 7.13.E-F-G-...-H). When Graham's Scan terminates, whatever that is left in S are the points of $CH(P)$ (see Figure 7.13.H, the stack contains (bottom) 11-0-1-4-7-10-11 (top)). Graham Scan's eliminates all the right turns! As three consecutive vertices in S always make left turns, we have a convex polygon.
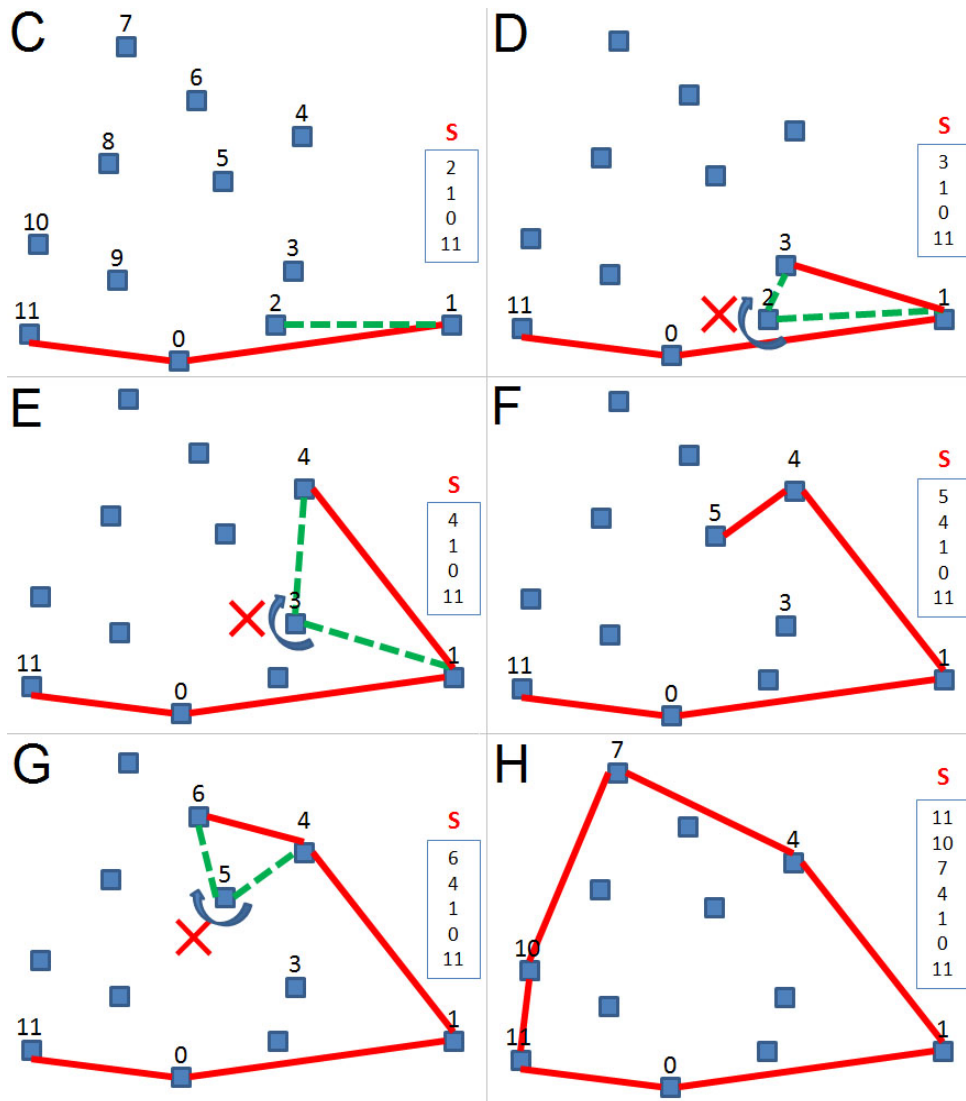
Figure 7.13: The Main Part of Graham's Scan algorithm

The implementation of Graham's Scan is shown below. We simply use a `vector<point> S` that behaves like a stack instead of using `stack<point> S`. The first part of Graham's Scan (finding the pivot) is just $O(n)$. The third part (the ccw tests) is also $O(n)$. This can be analyzed from the fact that each of the n vertices can only be pushed onto the stack once and popped from the stack once. The second part (sorts points by angle w.r.t pivot `P[0]`) is the *bulkiest* part that requires $O(n \log n)$. Overall, Graham's scan runs in $O(n \log n)$.

```
// continuation from the earlier part
  // third, the ccw tests
  vector<point> S;
  S.push_back(P[n-1]); S.push_back(P[0]); S.push_back(P[1]);   // initial S
  i = 2;                                          // then, we check the rest
  while (i < n) {          // note: N must be >= 3 for this method to work
    j = (int)S.size()-1;
    if (ccw(S[j-1], S[j], P[i])) S.push_back(P[i++]);  // left turn, accept
    else S.pop_back(); }   // or pop the top of S until we have a left turn
  return S; }                                     // return the result
```

We end this section and this chapter by pointing readers to another visualization tool, this time the visualization of several convex hull algorithms, including Graham's Scan, Andrew's Monotone Chain algorithm (see **Exercise 7.3.7.4***), and Jarvis's March algorithm. We also encourage readers to explore our source code to solve various programming exercises listed in this section.

Visualization: `www.comp.nus.edu.sg/~stevenha/visualization/convexhull.html`

Source code: `ch7_04_polygon.cpp/java`

Exercise 7.3.7.1: Suppose we have 5 points, $P = \{(0,0), (1,0), (2,0), (2,2), (0,2))\}$. The convex hull of these 5 points are actually these 5 points themselves (plus one, as we loop back to vertex (0, 0)). However, our Graham's scan implementation removes point (1, 0) as (0, 0)-(1, 0)-(2, 0) are collinear. Which part of the Graham's scan implementation that we have to modify to accept collinear points?

Exercise 7.3.7.2: In function `angleCmp`, there is a call to function: `atan2`. This function is used to compare the two angles but what is actually returned by `atan2`? Investigate!

Exercise 7.3.7.3*: Test the Graham's Scan code above: `CH(P)` on these corner cases. What is the convex hull of:

1. A single point, e.g. $P_1 = \{(0,0)\}$?

2. Two points (a line), e.g. $P_2 = \{(0,0), (1,0)\}$?

3. Three points (a triangle), e.g. $P_3 = \{(0,0), (1,0), (1,1)\}$?

4. Three points (a collinear line), e.g. $P_4 = \{(0,0), (1,0), (2,0)\}$?

5. Four points (a collinear line), e.g. $P_5 = \{(0,0), (1,0), (2,0), (3,0)\}$?

Exercise 7.3.7.4*: The Graham's Scan implementation above can be inefficient for large n as `atan2` is recalculated every time an angle comparison is made (and it is quite problematic when the angle is close to 90 degrees). Actually, the same basic idea of Graham's Scan also works if the input is sorted based on x-coordinate (and in case of a tie, by y-coordinate) instead of angle. The hull is now computed in 2 steps producing the *upper* and *lower* parts of the hull. This modification was devised by A. M. Andrew and known as Andrew's Monotone Chain Algorithm. It has the same basic properties as Graham's Scan but avoids costly comparisons between angles [9]. Investigate this algorithm and implement it!

Below, we provide a list of programming exercises related to polygon. Without pre-written library code discussed in this section, many of these problems look 'hard'. With the library code, they become manageable as the problem can now be decomposed into a few library routines. Spend some time to attempt them, especially the **must try *** ones.

Programming Exercises related to Polygon:

1. UVa 00109 - Scud Busters (find `CH`, test if point `inPolygon`, `area` of polygon)

2. *UVa 00137 - Polygons* (convex polygon intersection, line segment intersection, `inPolygon`, `CH`, `area`, inclusion-exclusion principle)

3. UVa 00218 - Moth Eradication (find `CH`, `perimeter` of polygon)

4. UVa 00361 - Cops and Robbers (check if a point is inside `CH` of Cop/Robber; if a point *pt* is inside a convex hull, then there is definitely a triangle formed using three vertices of the convex hull that contains *pt*)

5. UVa 00478 - Points in Figures: ... (`inPolygon`/inTriangle; if the given polygon *P* is *convex*, there is another way to check if a point *pt* is inside or outside *P* other than the way mentioned in this section; we can triangulate *P* into triangles with *pt* as one of the vertex, then sum the areas of the triangles; if it is the same as the area of polygon *P*, then *pt* is inside *P*; if it is larger, then *pt* is outside *P*)

6. *UVa 00596 - The Incredible Hull* (`CH`, output formatting is a bit tedious)

7. UVa 00634 - Polygon (`inPolygon`; the polygon can be convex or concave)

8. UVa 00681 - Convex Hull Finding (pure `CH` problem)

9. UVa 00858 - Berry Picking (ver line-polygon intersect; sort; alternating segments)

10. **_UVa 01111 - Trash Removal *_** (LA 5138, World Finals Orlando11, `CH`, distance of each `CH` side—which is parallel to the side—to each vertex of the `CH`)

11. UVa 01206 - Boundary Points (LA 3169, Manila06, convex hull `CH`)

12. *UVa 10002 - Center of Mass?* (centroid, center of `CH`, `area` of polygon)

13. UVa 10060 - A Hole to Catch a Man (`area` of polygon)

14. UVa 10065 - Useless Tile Packers (find `CH`, `area` of polygon)

15. UVa 10112 - Myacm Triangles (test if point `inPolygon`/inTriangle, see UVa 478)

16. UVa 10406 - Cutting tabletops (vector, `rotate`, `translate`, then `cutPolygon`)

17. **UVa 10652 - Board Wrapping *** (`rotate`, `translate`, `CH`, `area`)

18. UVa 11096 - Nails (very classic `CH` problem, start from here)

19. **UVa 11265 - The Sultan's Problem *** (`cutPolygon`, `inPolygon`, `area`)

20. UVa 11447 - Reservoir Logs (`area` of polygon)

21. UVa 11473 - Campus Roads (`perimeter` of polygon)

22. UVa 11626 - Convex Hull (find `CH`, be careful with collinear points)

7.4 Solution to Non-Starred Exercises

Exercise 7.2.1.1: 5.0.

Exercise 7.2.1.2: (-3.0, 10.0).

Exercise 7.2.1.3: (-0.674, 10.419).

Exercise 7.2.2.1: The line equation $y = mx + c$ cannot handle all cases: Vertical lines has 'infinite' gradient/slope in this equation and 'near vertical' lines are also problematic. If we use this line equation, we have to treat vertical lines separately in our code which decreases the probability of acceptance. Fortunately, this can be avoided by using the better line equation $ax + by + c = 0$.

Exercise 7.2.2.2: -0.5 * x + 1.0 * y - 1.0 = 0.0

Exercise 7.2.2.3: 1.0 * x + 0.0 * y - 2.0 = 0.0. If you use the $y = mx + c$ line equation, you will have $x = 2.0$ instead, but you cannot represent a vertical line using this form $y = ?$.

Exercise 7.2.2.4: Given 2 points $(x1, y1)$ and $(x2, y2)$, the slope can be calculated with $m = (y2 - y1)/(x2 - x1)$. Subsequently the y-intercept c can be computed from the equation by substitution of the values of a point (either one) and the line gradient m. The code will looks like this. See that we have to deal with vertical line separately and awkwardly.

```
struct line2 { double m, c; };        // another way to represent a line

int pointsToLine2(point p1, point p2, line2 &l) {
 if (p1.x == p2.x) {                        // special case: vertical line
   l.m = INF;                       // l contains m = INF and c = x_value
   l.c = p1.x;                      // to denote vertical line x = x_value
   return 0;    // we need this return variable to differentiate result
 }
 else {
   l.m = (double)(p1.y - p2.y) / (p1.x - p2.x);
   l.c = p1.y - l.m * p1.x;
   return 1;      // l contains m and c of the line equation y = mx + c
} }
```

Exercise 7.2.2.5:

```
// convert point and gradient/slope to line
void pointSlopeToLine(point p, double m, line &l) {
  l.a = -m;                                              // always -m
  l.b = 1;                                               // always 1
  l.c = -((l.a * p.x) + (l.b * p.y)); }           // compute this
```

Exercise 7.2.2.6: (5.0, 3.0).

Exercise 7.2.2.7: (4.0, 2.5).

Exercise 7.2.2.8: (-3.0, 5.0).

Exercise 7.2.2.9: (0.0, 4.0). The result is different from **Exercise 7.2.2.8**. 'Translate then Rotate' is different from 'Rotate then Translate'. Be careful in sequencing them.

Exercise 7.2.2.10: (1.0, 2.0). If the rotation center is not origin, we need to translate the input point c (3, 2) by a vector described by $-p$, i.e. (-2, -1) to point c' (1, 1). Then, we perform the 90 degrees counter clockwise rotation around origin to get c'' (-1, 1). Finally, we translate c'' to the final answer by a vector described by p to point (1, 2).

Exercise 7.2.2.11: The solution is shown below:

```
void closestPoint(line l, point p, point &ans) {
  line perpendicular;          // perpendicular to l and pass through p
  if (fabs(l.b) < EPS) {                // special case 1: vertical line
    ans.x = -(l.c);    ans.y = p.y;      return; }

  if (fabs(l.a) < EPS) {                // special case 2: horizontal line
    ans.x = p.x;        ans.y = -(l.c);   return; }

  pointSlopeToLine(p, 1 / l.a, perpendicular);         // normal line
  // intersect line l with this perpendicular line
  // the intersection point is the closest point
  areIntersect(l, perpendicular, ans); }
```

Exercise 7.2.2.12: The solution is shown below. Other solution exists:

```
// returns the reflection of point on a line
void reflectionPoint(line l, point p, point &ans) {
  point b;
  closestPoint(l, p, b);                         // similar to distToLine
  vec v = toVector(p, b);                          // create a vector
  ans = translate(translate(p, v), v); }        // translate p twice
```

Exercise 7.2.2.13: 63.43 degrees.

Exercise 7.2.2.14: Point p (3,7) → point q (11,13) → point r (35,30) form a right turn. Therefore, point p is on the right side of a line that passes through point p and point r. Note: If point r is at (35, 31), then p, q, r are collinear.

Exercise 7.2.3.1: See Figure 7.14 below.
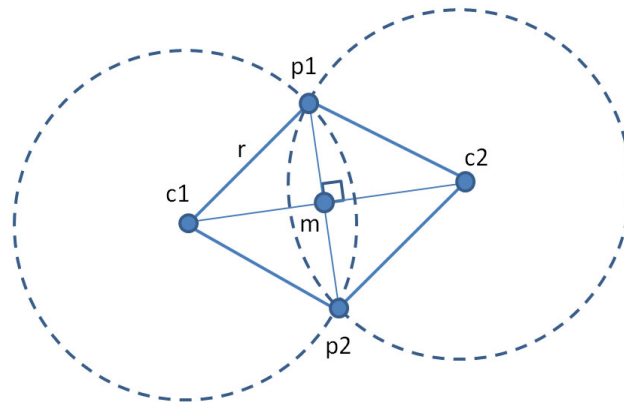
Figure 7.14: Explanation for Circle Through 2 Points and Radius

Let $c1$ and $c2$ be the centers of the 2 possible circles that go through 2 given points $p1$ and $p2$ and have radius r. The quadrilateral $p1 - c2 - p2 - c1$ is a rhombus, since its four sides are equal. Let m be the intersection of the 2 diagonals of the rhombus $p1 - c2 - p2 - c1$. According to the property of a rhombus, m bisects the 2 diagonals, and the 2 diagonals are perpendicular to each other. We realize that $c1$ and $c2$ can be calculated by scaling the vectors $mp1$ and $mp2$ by an appropriate ratio $(mc1/mp1)$ to get the same magnitude as $mc1$, then rotating the points $p1$ and $p2$ around m by 90 degrees. In the implementation given in **Exercise 7.2.3.1**, the variable h is *half* the ratio $mc1/mp1$ (one can work out on paper why h can be calculated as such). In the 2 lines calculating the coordinates of one of the centers, the first operands of the additions are the coordinates of m, while the second operands of the additions are the result of scaling and rotating the vector $mp2$ around m.

Exercise 7.2.4.1: We can use double data type that has larger range. However, to further reduce the chance of overflow, we can rewrite the Heron's formula into $A = sqrt(s) \times sqrt(s - a) \times sqrt(s - b) \times sqrt(s - c))$. However, the result will be slightly less precise as we call *sqrt* 4 times instead of once.

Exercise 7.3.6.1: Swap point a and b when calling `cutPolygon(a, b, Q)`.

Exercise 7.3.7.1: Edit the `ccw` function to accept collinear points.

Exercise 7.3.7.2: The function `atan2` computes the inverse tangent of $\frac{y}{x}$ using the signs of arguments to correctly determine quadrant.

7.5 Chapter Notes

Some material in this chapter are derived from the material courtesy of **Dr Cheng Holun, Alan** from School of Computing, National University of Singapore. Some library functions are customized from **Igor Naverniouk**'s library: `http://shygypsy.com/tools/`.

Compared to the first edition of this book, this chapter has, just like Chapter 5 and 6, grown to about twice its original size. However, the material mentioned here are still far from complete, especially for ICPC contestants. If you are preparing for ICPC, it is a good idea to dedicate one person in your team to study this topic in depth. This person should master basic geometry formulas and advanced computational geometry techniques, perhaps by reading relevant chapters in the following books: [50, 9, 7]. But not just the theory, he must also train himself to code *robust* geometry solutions that are able to handle degenerate (special) cases and precision errors.

The other computational geometry techniques that have not been discussed yet in this chapter are the **plane sweep** technique, intersection of **other geometric objects** including line segment-line segment intersection, various Divide and Conquer solutions for several classical geometry problems: **The Closest Pair Problem**, **The Furthest Pair Problem**, **Rotating Calipers** algorithm, etc. Some of these problems are discussed in Chapter 9.

Statistics	First Edition	Second Edition	Third Edition
Number of Pages	13	22 (+69%)	29 (+32%)
Written Exercises	-	20	22+9*=31 (+55%)
Programming Exercises	96	103 (+7%)	96 (-7%)

The breakdown of the number[16] of programming exercises from each section is shown below:

Section	Title	Appearance	% in Chapter	% in Book
7.2	**Basic Geometry Objects ...**	74	77%	4%
7.3	**Algorithm on Polygon ...**	22	23%	1%

---

[16]The total decreases a bit although we have added several new problems because some of the problems are moved to Chapter 8

L-R: Mr Raymond, Felix, Andrian, Andoko @ ACM ICPC World Finals, Tokyo 2007

Chapter 8

More Advanced Topics

Genius is one percent inspiration, ninety-nine percent perspiration.
— **Thomas Alva Edison**

8.1 Overview and Motivation

The main purpose of having this chapter is organizational. The first two sections of this chapter contain the harder material from Chapter 3 and 4. In Section 8.2 and 8.3, we discuss the more challenging variants and techniques involving the two most popular problem solving paradigms: Complete Search and Dynamic Programming. Putting these material in the earlier chapters will probably scare off some new readers of this book.

Section 8.4 contains discussions of complex problems that require *more than one* algorithms and/or data structures. These discussions can be confusing for new programmers if they are listed in the earlier chapters. It is more appropriate to discuss them in this chapter, after various (easier) data structures and algorithms have been discussed. Therefore, it is a good idea to read Chapter 1-7 first before reading this section.

We also encourage readers to avoid rote memorization of the solutions but more importantly, please try to understand the key ideas that may be applicable to other problems.

8.2 More Advanced Search Techniques

In Section 3.2, we have discussed various (simpler) iterative and recursive (backtracking) Complete Search techniques. However, some harder problems require *more clever* Complete Search solutions to avoid the Time Limit Exceeded (TLE) verdict. In this section, we discuss some of these techniques with several examples.

8.2.1 Backtracking with Bitmask

In Section 2.2, we have seen that bitmask can be used to model a small set of Boolean. Bitmask operations are very lightweight and therefore every time we need to use a small set of Boolean, we can consider using bitmask technique to speed up our (Complete Search) solution. In this subsection, we give two examples.

The N-Queens Problem, Revisited

In Section 3.2.2, we have discussed UVa 11195 - Another n-Queen Problem. But even after we have improved the left and right diagonal checks by storing the availability of each of the

n rows and the $2 \times n - 1$ left/right diagonals in three `bitset`s, we still get TLE. Converting these three `bitset`s into three bitmasks help a bit, but this is still TLE.

Fortunately, there is a better way to use these row, left diagonal, and right diagonal checks, as described below. This formulation[1] allows for efficient backtracking with bitmask. We will straightforwardly use three bitmasks for `rw`, `ld`, and `rd` to represent the state of the search. The on bits in bitmasks `rw`, `ld`, and `rd` describe which *rows* are attacked in the *next column*, due to *row*, *left diagonal*, or *right diagonal* attacks from previously placed queens, respectively. Since we consider one column at a time, there will only be n possible left/right diagonals, hence we can have three bitmasks of the same length of n bits (compared with $2 \times n - 1$ bits for the left/right diagonals in the earlier formulation in Section 3.2.2).

Notice that although both solutions (the one in Section 3.2.2 and the one above) use the same data structure: Three bitmasks, the one described above is much more efficient. This highlights the need for problem solver to think from various angles.

We first show the short code of this recursive backtracking with bitmask for the (general) n-queens problem with $n = 5$ and then explain how it works.

```
int ans = 0, OK = (1 << 5) - 1;                    // testing for n = 5 queens

void backtrack(int rw, int ld, int rd) {
  if (rw == OK) { ans++; return; }            // if all bits in 'rw' are on
  int pos = OK & (~(rw | ld | rd));        // the '1's in 'pos' are available
  while (pos) {                                // this loop is faster than O(n)
    int p = pos & -pos;              // Least Significant One---this is fast
    pos -= p;                                         // turn off that on bit
    backtrack(rw | p, (ld | p) << 1, (rd | p) >> 1);             // clever
} }

int main() {
  backtrack(0, 0, 0);                                    // the starting point
  printf("%d\n", ans);                       // the answer should be 10 for n = 5
} // return 0;
```

For $n = 5$, we start with state (`rw`, `ld`, `rd`) = (0, 0, 0) = $(00000, 00000, 00000)_2$. This state is shown in Figure 8.1. The variable `OK` = (1 << 5) - 1 = $(11111)_2$ is used both as terminating condition check and to help decide which rows are available for a certain column. The operation `pos = OK & (~(rw | ld | rd))` *combines* the information of which rows in the next column are attacked by the previously placed queens (via row, left diagonal, or right diagonal attacks), *negates* the result, and *combines* it with `OK` to yield the rows that are *available* for the next column. Initially, all rows in column 0 are available.
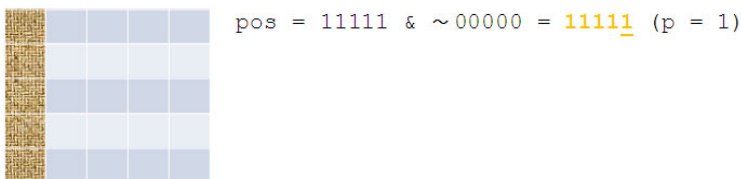
Figure 8.1: 5 Queens problem: The initial state

---

[1] While this solution is customized for this N-Queens problem, we can probably use parts of the solution for another problem.

Complete Search (the recursive backtracking) will try all possible rows (that is, all the *on bits* in variable `pos`) of a certain column one by one. Previously in Section 3.2.1, we have seen a way to explore all the on bits of a bitmask in $O(n)$:

```
for (p = 0; p < n; p++)                                        // O(n)
   if (pos && (1 << p))                    // if this bit 'p' is on in 'pos'
      // process p
```

However, this is not the most efficient way. As the recursive backtracking goes deeper, less and less rows are available for selection. Instead of trying all n rows, we can speed up the loop above by just trying all the on bits in variable `pos`. The loop below runs in $O(k)$:

```
while (pos) { // O(k), where k is the number of bits that are on in 'pos'
   int p = pos & -pos;     // determine the Least Significant One in 'pos'
   pos -= p;                              // turn off that on bit
   // process p
}
```

Back to our discussion, for `pos = (11111)`$_2$, we will first start with `p = pos & -pos = 1`, or row 0. After placing the first queen (queen 0) at row 0 of column 0, row 0 is no longer available for the next column 1 and this is quickly captured by bit operation `rw | p` (and also `ld | p` and `rd | p`). Now here is the beauty of this solution. A left/right diagonal increases/decreases the row number that it attacks by one as it changes to the next column, respectively. A shift left/right operation: `(ld | p) << 1` and `(rd | p) >> 1` can nicely capture these behaviour effectively. In Figure 8.2, we see that for the next column 1, row 1 is not available due to left diagonal attack by queen 0. Now only row 2, 3, and 4 are available for column 1. We will start with row 2.


```
rw                    = 00001 (1)
ld = 00001 << 1 = 00010 (2)
rd = 00001 >> 1 = 00000|1 (0, the LSB is removed)
---------------------- OR
                  00011 -> NEGATE -> 11100
pos = 11111 & 11100 = 11100 (p = 4)
```
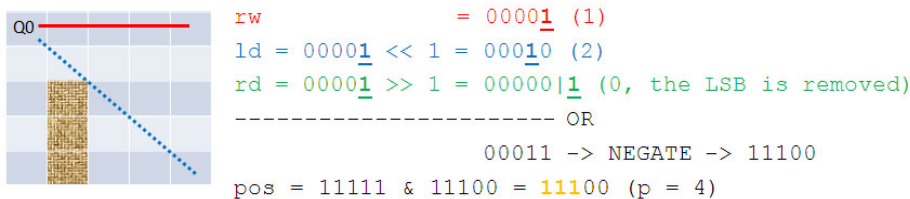
Figure 8.2: 5 Queens problem: After placing the first queen

After placing the second queen (queen 1) at row 2 of column 1, row 0 (due to queen 0) and now row 2 are no longer available for the next column 2. The shift left operation for the left diagonal constraint causes row 2 (due to queen 0) and now row 3 to be unavailable for the next column 2. The shift right operation for the right diagonal constraint causes row 1 to be unavailable for the next column 2. Therefore, only row 4 is available for the next column 2 and we have to choose it next (see Figure 8.3).


```
rw                    = 00101 (5)
ld = 00110 << 1 = 01100 (12)
rd = 00100 >> 1 = 00010 (2)
---------------------- OR
                  01111 -> NEGATE -> 10000
pos = 11111 & 10000 = 10000 (p = 16)
```
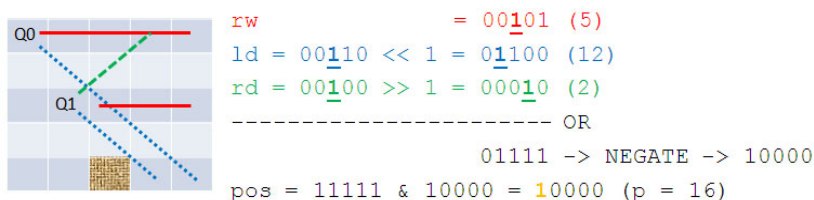
Figure 8.3: 5 Queens problem: After placing the second queen

301

After placing the third queen (queen 2) at row 4 of column 2, row 0 (due to queen 0), row 2 (due to queen 1), and now row 4 are no longer available for the next column 3. The shift left operation for the left diagonal constraint causes row 3 (due to queen 0) and row 4 (due to queen 1) to be unavailable for the next column 3 (there is no row 5—the MSB in bitmask `ld` is unused). The shift right operation for the right diagonal constraint causes row 0 (due to queen 1) and now row 3 to be unavailable for the next column 3. Combining all these information, only row 1 is available for the next column 3 and we have to choose it next (see Figure 8.4).


```
rw              =  10101 (21)
ld = 11100 << 1 = 111000 (56, but the MSB is unused)
rd = 10010 >> 1 =  01001 (9)
----------------------- OR
                 111101 -> NEGATE -> 000010
pos = 11111 & 000010 = 00010 (p = 2)
```
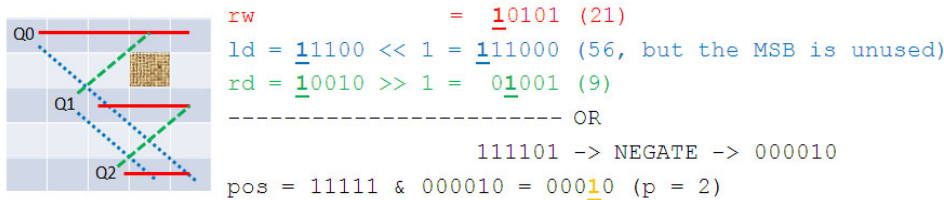
Figure 8.4: 5 Queens problem: After placing the third queen

The same explanation is applicable for the fourth and the fifth queen (queen 3 and 4) as shown in Figure 8.5. We can continue this process to get the other 9 solutions for $n = 5$.


```
rw              =   10111 (23)
ld = 111010 << 1 = 1110100 (116, two MSBs are unused)
rd =  01011 >> 1 =   00101 (5)
------------------------- OR
                  1110111 -> NEGATE -> 0001000
pos = 11111 & 0001000 = 01000 (p = 8)
```

```
rw              =  11111 (we get one solution)
```
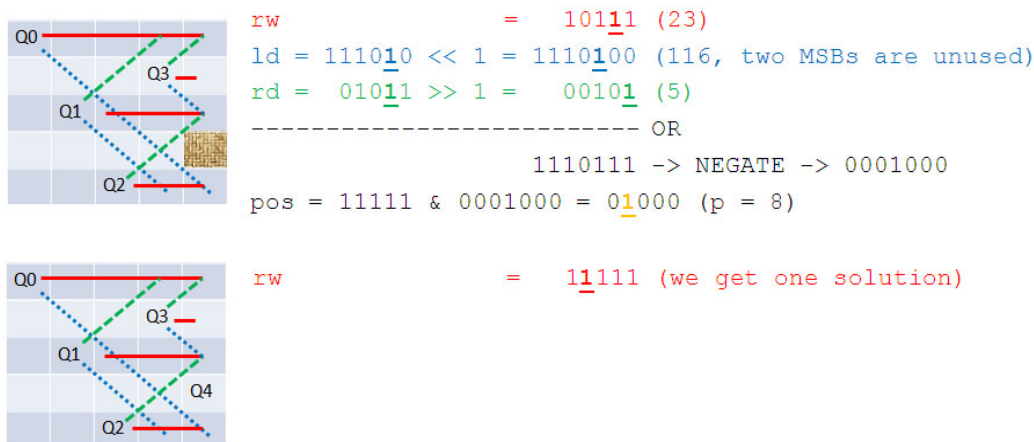
Figure 8.5: N-Queens, after placing the fourth and the fifth queens

With this technique, we can solve UVa 11195. We just need to modify the given code above to take the bad cells—which can also be modeled as bitmasks—into consideration. Let's roughly analyze the worst case for $n \times n$ board with no bad cell. Assuming that this recursive backtracking with bitmask has approximately two less rows available at each step, we have a time complexity of $O(n!!)$ where $n!!$ is a notation of multifactorial. For $n = 14$ with no bad cell, the recursive backtracking solution in Section 3.2.2 requires up to $O(14!) \approx 87178M$ operations whereas the recursive backtracking with bitmask above only require around $O(14!!) = 14 \times 12 \times 10 \times \ldots \times 2 = 645120$ operations.

Compact Adjacency Matrix Graph Data Structure

The UVa 11065 - Gentlemen Agreement problem boils down to computation of two integers: The number of Independent Set and the size of the Maximum Independent Set (MIS—see Section 4.7.4 for the problem definition) of a given *general* graph with $V \leq 60$. Finding the MIS of a general graph is an NP-hard problem. Therefore, there is no hope for a polynomial algorithm for this problem.

One solution is the following clever recursive backtracking. The state of the search is a triple: $(i, used, depth)$. The first parameter i implies that we can consider vertices in $[i..V\text{-}1]$ to be included in the Independent Set. The second parameter $used$ is a bitmask of length V bits that denotes which vertices are no longer available to be used anymore for the current Independent Set because at least one of their neighbors have been included in the Independent Set. The third parameter $depth$ stores the depth of the recursion—which is also the size of the current Independent Set.

There is a clever bitmask trick for this problem that can be used to speed up the solution significantly. Notice that the input graph is small, $V \leq 60$. Therefore, we can store the input graph in an Adjacency Matrix of size up to $V \times V$ (for this problem, we set all cells along the main diagonal of the Adjacency Matrix to true). However, we can compress *one row* of V Booleans ($V \leq 60$) into one bitmask using a 64-bit signed integer.

With this compact Adjacency Matrix $AdjMat$—which is just V rows of 64-bit signed integers—we can use a fast bitmask operation to flag neighbors of vertices efficiently. If we decide to take a free vertex i—i.e. $(used \ \& \ (1 << i)) == 0$, we increase $depth$ by one and then use an $O(1)$ bitmask operation: $used \ | \ AdjMat[i]$ to flag *all* neighbors of i including itself (remember that $AdjMat[i]$ is also a bitmask of length V bits with the i-th bit on).

When all bits in bitmask $used$ is turned on, we have just found one more Independent Set. We also record the largest $depth$ value throughout the process as this is the size of the Maximum Independent Set of the input graph. The key parts of the code is shown below:

```
void rec(int i, long long used, int depth) {
  if (used == (1 << V) - 1) {              // all intersection are visited
    nS++;                                       // one more possible set
    mxS = max(mxS, depth);                         // size of the set
  }
  else {
    for (int j = i; j < V; j++)
      if (!(used & (1 << j)))          // if intersection j is not yet used
        rec(j + 1, used | AdjMat[j], depth + 1);      // fast bit operation
  }
}

// inside int main()
  // a more powerful, bit-wise adjacency list (for faster set operations)
  for (int i = 0; i < V; i++)
    AdjMat[i] = (1 << i);                                  // i to itself
  for (int i = 0; i < E; i++) {
    scanf("%d %d", &a, &b);
    AdjMat[a] |= (1 << b);
    AdjMat[b] |= (1 << a);
  }
```

Exercise 8.2.1.1*: Sudoku puzzle is another NP-complete problem. The recursive backtracking to find one solution for a standard 9×9 ($n = 3$) Sudoku board can be speed up using bitmask. For each empty cell (r, c), we try putting a digit $[1..n^2]$ one by one if it is a valid move. The n^2 row, n^2 column, and $n \times n$ square checks can be done with three bitmasks of length n^2 bits. Solve two similar problems: UVa 989 and UVa 10957 with this technique!

8.2.2   Backtracking with Heavy Pruning

Problem I - 'Robots on Ice' in ACM ICPC World Finals 2010 can be viewed as a 'tough test on pruning strategy'. The problem description is simple: Given an $M \times N$ board with 3 check-in points {A, B, C}, find a Hamiltonian[2] path of length $(M \times N)$ from coordinate (0, 0) to coordinate (0, 1). This Hamiltonian path must hit the three check points: A, B, and C at one-quarter, one-half, and three-quarters of the way through its path, respectively. Constraints: $2 \le M, N \le 8$.

Example: If given the following 3×6 board with A = (row, col) = (2, 1), B = (2, 4), and C = (0, 4) as in Figure 8.6, then we have two possible paths.
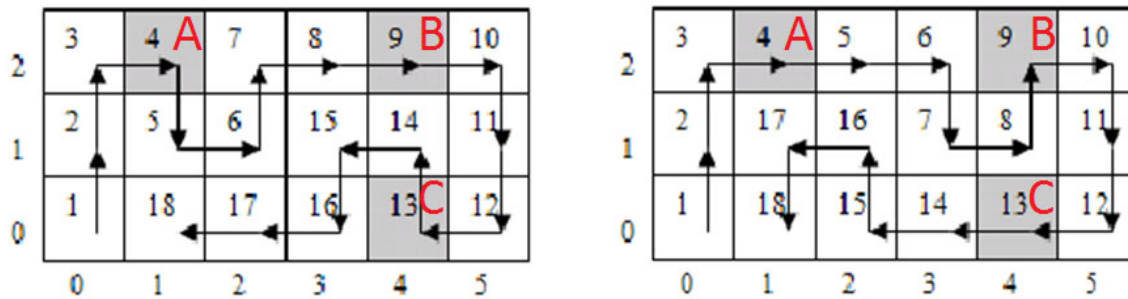
Figure 8.6: Visualization of UVa 1098 - Robots on Ice

A naïve recursive backtracking algorithm will get TLE as there are 4 choices at every step and the maximum path length is $8 \times 8 = 64$ in the largest test case. Trying all 4^{64} possible paths is infeasible. To speed up the algorithm, we must prune the search space if the search:

1. Wanders outside the $M \times N$ grid (obvious),

2. Does not hit the appropriate target check point at 1/4, 1/2, or 3/4 distance—the presence of these three check points actually *reduce* the search space,

3. Hits target check point earlier than the target time,

4. Will not be able to reach the next check point on time from the current position,

5. Will not be able to reach certain coordinates as the current partial path self-block the access to those coordinates. This can be checked with a simple DFS/BFS (see Section 4.2). First, we run DFS/BFS from the goal coordinate (0, 1). If there are coordinates in the $M \times N$ grid that are *not* reachable from (0, 1) and *not yet visited* by the current partial path, we can prune the current partial path.

Exercise 8.2.2.1*: The five pruning strategies mentioned in this subsection are good but actually insufficient to pass the time limit set for LA 4793 and UVa 1098. There is a faster solution for this problem that utilizes the meet in the middle technique (see Section 8.2.4). This example illustrates that the choice of time limit setting may determine which Complete Search solutions are considered as fast enough. Study the idea of meet in the middle technique in Section 8.2.4 and apply it to solve this Robots on Ice problem.

---

[2]A Hamiltonian path is a path in an undirected graph that visits each vertex exactly once.

8.2.3 State-Space Search with BFS or Dijkstra's

In Section 4.2.2 and 4.4.3, we have discussed two standard graph algorithms for solving the Single-Source Shortest Paths (SSSP) problem. BFS can be used if the graph is unweighted while Dijkstra's should be used if the graph is weighted. The SSSP problems listed in Chapter 4 are still easier in the sense that most of the time we can easily see 'the graph' in the problem description. This is no longer true for some harder graph searching problems listed in this section where the (usually implicit) graphs are no longer trivial to see and the state/vertex can be a complex object. In such case, we usually name the search as 'State-Space Search' instead of SSSP.

When the state is a complex object—e.g. a pair (position, bitmask) in UVa 321 - The New Villa, a quad (row, col, direction, color) in UVa 10047 - The Monocycle, etc—, we normally do not use vector<int> dist to store the distance information as in the standard BFS/Dijkstra's implementation. This is because such state may not be easily converted into integer indices. One solution is to use map<VERTEX-TYPE, int> dist instead. This trick adds a (small) $\log V$ factor to the time complexity of BFS/Dijkstra's. But for complex State-Space Search, this extra runtime overhead may be acceptable in order to bring down the overall coding complexity. In this subsection, we show one example of such complex State-Space Search.

Exercise 8.2.3.1: How to store VERTEX-TYPE if it is a pair, a triple, or a quad of information in both C++ and Java?

Exercise 8.2.3.2: Similar question as in **Exercise 8.2.3.1**, but VERTEX-TYPE is a much more complex object, e.g. an array.

Exercise 8.2.3.3: Is it possible that State-Space Search is cast as a maximization problem?

UVa 11212 - Editing a Book

Abridged Problem Description: Given n paragraphs numbered from 1 to n, arrange them in the order of 1, 2, ..., n. With the help of a clipboard, you can press Ctrl-X (cut) and Ctrl-V (paste) several times. You cannot cut twice before pasting, but you can cut several contiguous paragraphs at the same time and these paragraphs will later be pasted in order. What is the minimum number of steps required?

Example 1: In order to make {2, 4, (1), 5, 3, 6} sorted, we cut paragraph (1) and paste it before paragraph 2 to have {1, 2, 4, 5, (3), 6}. Then, we cut paragraph (3) and paste it before paragraph 4 to have {1, 2, 3, 4, 5, 6}. The answer is two steps.

Example 2: In order to make {(3, 4, 5), 1, 2} sorted, we cut three paragraphs at the same time: (3, 4, 5) and paste them after paragraph 2 to have {1, 2, 3, 4, 5}. This is just one single step. This solution is not unique as we can have the following alternative answer: We cut two paragraphs at the same time: (1, 2) and paste them before paragraph 3 to get {1, 2, 3, 4, 5}—this is also one single step.

The loose upper bound of the number of steps required to rearrange these n paragraphs is $O(k)$, where k is the number of paragraphs that are initially in the wrong positions. This is because we can use the following 'trivial' algorithm (which is incorrect): Cut a single paragraph that is in the wrong position and paste that paragraph in the correct position. After k such cut-paste operation, we will definitely have a sorted paragraph. But this may not be the shortest way.

For example, the 'trivial' algorithm above will process {5, 4, 3, 2, 1} as follows:
{(5), 4, 3, 2, 1} → {(4), 3, 2, 1, 5} → {(3), 2, 1, 4, 5} → {(2), 1, 3, 4, 5} → {1, 2, 3, 4, 5}
of total 4 cut-paste steps. This is not optimal, as we can solve this instance in only 3 steps:
{5, 4, (3, 2), 1} → {3, (2, 5), 4, 1} → {3, 4, (1, 2), 5} → {1, 2, 3, 4, 5}.

This problem has a *huge* search space that even for an instance with small $n = 9$, it is near impossible for us to get the answer manually, e.g. We likely will not start drawing the recursion tree just to verify that we need at least 4 steps to sort {5, 4, 9, 8, 7, 3, 2, 1, 6} and at least 5 steps to sort {9, 8, 7, 6, 5, 4, 3, 2, 1}.

The state of this problem is a *permutation* of paragraphs. There are at most $O(n!)$ permutations of paragraphs. With maximum $n = 9$ in the problem statement, this is 9! or 362880. So, the number of vertices of the State-Space graph is not that big actually.

The difficulty of this problem lies in the number of *edges* of the State-Space graph. Given a permutation of length n (a vertex), there are $_nC_2$ possible cutting points (index $i, j \in [1..n]$) and there are n possible pasting points (index $k \in [1..(n - (j - i + 1))]$). Therefore, for each of the $O(n!)$ vertex, there are about $O(n^3)$ edges connected to it.

The problem actually asked for the shortest path from the source vertex/state (the input permutation) to the destination vertex (a sorted permutation) on this unweighted but huge State-Space graph. The worst case behavior if we run a single $O(V + E)$ BFS on this State-Space graph is $O(n! + (n! * n^3)) = O(n! * n^3)$. For $n = 9$, this is $9! * 93 = 264539520 \approx 265M$ operations. This solution most likely will receive a TLE (or maybe MLE) verdict.

We need a better solution, which we will see in the next Section 8.2.4.

8.2.4 Meet in the Middle (Bidirectional Search)

For some SSSP (but usually State-Space Search) problems on huge graph and we know two vertices: The source vertex/state s and the destination vertex/state t, we may be able to *significantly* reduce the time complexity of the search by searching from *both directions* and hoping that the search will *meet in the middle*. We illustrate this technique by continuing our discussion of the hard UVa 11212 problem.

Before we continue, we need to make a remark that the meet in the middle technique does not always refer to bidirectional BFS. It is a problem solving strategy of 'searching from two directions/parts' that may appear in another form in other difficult searching problem, e.g. see **Exercise 3.2.1.4***.

UVa 11212 - Editing a Book (Revisited)

Although the worst case time complexity of the State-Space Search of this problem is bad, the largest possible answer for this problem is small. When we run BFS on the largest test case with $n = 9$ from the destination state t (the sorted permutation {1, 2, ..., 9}) to reach all other states, we find out that for this problem, the maximum depth of the BFS for $n = 9$ is just 5 (after running it for *a few minutes*—which is TLE in contest environment).

This important information allows us to perform bidirectional BFS by choosing only to go to depth 2 from each direction. While this information is not a necessary condition for us to run a bidirectional BFS, it can help reducing the search space.

There are three possible cases which we discuss below.

Case 1: Vertex s is within two steps away from vertex t (see Figure 8.7).
We first run BFS (max depth of BFS = 2) from the target vertex t to populate distance information from t: `dist_t`. If the source vertex s is already found, i.e. `dist_t[s]` is not `INF`, then we return this value. The possible answers are: 0 (if $s = t$), 1, or 2 steps.
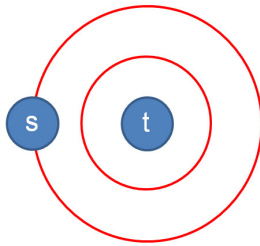
Figure 8.7: Case 1: Example when s is two steps away from t

Case 2: Vertex s is within three to four steps away from vertex t (see Figure 8.8).
If we do not manage to find the source vertex s after Case 1 above, i.e. `dist_t[s] = INF`,
we know that s is located further away from vertex t. We now run BFS from the source
vertex s (also with max depth of BFS $= 2$) to populate distance information from s: `dist_s`.
If we encounter a common vertex v 'in the middle' during the execution of this second BFS,
we know that vertex v is within two layers away from vertex t and s. The answer is therefore
`dist_s[v] + dist_t[v]` steps. The possible answers are: 3 or 4 steps.
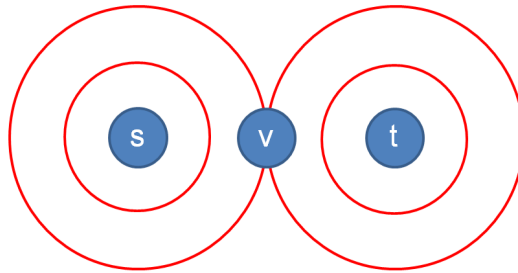
Figure 8.8: Case 2: Example when s is four steps away from t

Case 3: Vertex s is exactly five steps away from vertex t (see Figure 8.9).
If we do not manage to find any common vertex v after running the second BFS in Case
2 above, then the answer is clearly 5 steps that we know earlier as s and t must always be
reachable. Stopping at depth 2 allows us to skip computing depth 3, which is *much more
time consuming* than computing depth 2.
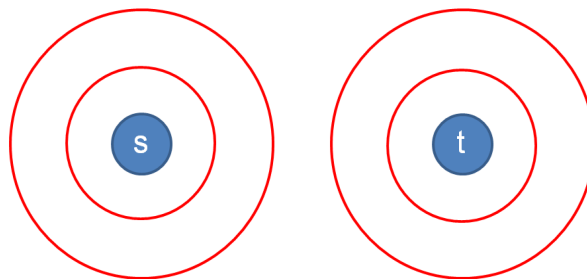
Figure 8.9: Case 3: Example when s is five steps away from t

We have seen that given a permutation of length n (a vertex), there are about $O(n^3)$ branches
in this huge State-Space graph. However, if we just run each BFS with at most depth 2, we
only execute at most $O((n^3)^2) = O(n^6)$ operations per BFS. With $n = 9$, this is $9^6 = 531441$
operations (this is greater than 9! as there are some overlaps). As the destination vertex
t is unchanged throughout the State-Space search, we can compute the first BFS from
destination vertex t just once. Then we compute the second BFS from source vertex s per
query. Our BFS implementation will have an additional log factor due to the usage of table
data structure (e.g. `map`) to store `dist_t` and `dist_s`. This solution is now Accepted.

307

8.2.5 Informed Search: A* and IDA*

The Basics of A*

Complete Search algorithms that we have seen earlier in Chapter 3, 4, and the earlier subsections of this Section are 'uninformed', i.e. all possible states reachable from the current state are *equally good*. For some problems, we do have access to more information (hence the name 'informed search') and we can use the clever A* search that employs heuristic to 'guide' the search direction.

Figure 8.10: 15 Puzzle

We illustrate this A* search using a well-known 15-puzzle problem. There are 15 slide-able tiles in the puzzle, each with a number from 1 to 15 on it. These 15 tiles are packed into a 4×4 frame with one tile missing. The possible actions are to slide the tile adjacent to the missing tile to the position of that missing tile. Another way of viewing these actions is: "To slide the *blank tile* rightwards, upwards, leftwards, or downwards". The objective of this puzzle is to arrange the tiles so that they looks like Figure 8.10, the 'goal' state.

This seemingly small puzzle is a headache for various search algorithms due to its enormous search space. We can represent a state of this puzzle by listing the numbers of the tiles row by row, left to right into an array of 16 integers. For simplicity, we assign value 0 to the blank tile so the goal state is $\{1, 2, 3, \ldots, 14, 15, 0\}$. Given a state, there can be up to 4 reachable states depending on the position of the missing tile. There are 2/3/4 possible actions if the missing tile is at the 4 corners/8 non-corner sides/4 middle cells, respectively. This is a huge search space.

However, these states are not equally good. There is a nice heuristic for this problem that can help guiding the search algorithm, which is the sum of Manhattan[3] distances between each (non blank) tile in the current state and its location in the goal state. This heuristic gives the lower bound of steps to reach the goal state. By combining the cost so far (denoted by $g(s)$) and the heuristic value (denoted by $h(s)$) of a state s, we have a better idea on where to move next. We illustrate this with a puzzle with starting state A below:

$$A = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & \mathbf{0} \\ 13 & 14 & 15 & 12 \end{bmatrix}$$

$$B = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & \mathbf{0} \\ 9 & 10 & 11 & \mathbf{8} \\ 13 & 14 & 15 & 12 \end{bmatrix} \quad C = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & \mathbf{0} & \mathbf{11} \\ 13 & 14 & 15 & 12 \end{bmatrix} \quad D = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & \mathbf{12} \\ 13 & 14 & 15 & \mathbf{0} \end{bmatrix}$$

The cost of the starting state A is $g(s) = 0$, no move yet. There are three reachable states $\{B, C, D\}$ from this state A with $g(B) = g(C) = g(D) = 1$, i.e. one move. But these three states are *not* equally good:

1. The heuristic value if we slide tile 0 upwards is $h(B) = 2$ as tile 8 and tile 12 are both off by 1. This causes $g(B) + h(B) = 1 + 2 = 3$.

2. The heuristic value if we slide tile 0 leftwards is $h(C) = 2$ as tile 11 and tile 12 are both off by 1. This causes $g(C) + h(C) = 1 + 2 = 3$.

3. But if we slide tile 0 downwards, we have $h(D) = 0$ as all tiles are in their correct position. This causes $g(D) + h(D) = 1 + 0 = 1$, the lowest combination.

---

[3]The Manhattan distance between two points is the sum of the absolute differences of their coordinates.

If we visit the states in ascending order of $g(s) + h(s)$ values, we will explore the states with the smaller expected cost first, i.e. state D in this example—which is the goal state. This is the essence of the A* search algorithm.

We usually implement this states ordering with the help of a priority queue—which makes the implementation of A* search very similar with the implementation of Dijkstra's algorithm presented in Section 4.4. Note that if $h(s)$ is set to 0 for all states, A* *degenerates* to Dijkstra's algorithm again.

As long as the heuristic function $h(s)$ never overestimates the true distance to the goal state (also known as **admissible heuristic**), this A* search algorithm is optimal. The hardest part in solving search problems using A* search is in finding such heuristic.

Limitations of A*

The problem with A* (and also BFS and Dijkstra's algorithms when used on large State-Space graph) that uses (priority) queue is that the memory requirement can be very huge when the goal state is far from the initial state. For some difficult searching problem, we may have to resort to the following related techniques.

Depth Limited Search

In Section 3.2.2, we have seen recursive backtracking algorithm. The main problem with pure backtracking is this: It may be trapped in an exploration of a very deep path that will not lead to the solution before eventually backtracks after wasting precious runtime.

Depth Limited Search (DLS) places a limit on how deep a backtracking can go. DLS stops going deeper when the depth of the search is longer than what we have defined. If the limit happens to be equal to the depth of the shallowest goal state, then DLS is faster than the general backtracking routine. However, if the limit is too small, then the goal state will be unreachable. If the problem says that the goal state is 'at most d steps away' from the initial state, then use DLS instead of general backtracking routine.

Iterative Deepening Search

If DLS is used wrongly, then the goal state will be unreachable although we have a solution. DLS is usually not used alone, but as part of Iterative Deepening Search (IDS).

IDS calls DLS with *increasing limit* until the goal state is found. IDS is therefore complete and optimal. IDS is a nice strategy that sidesteps the problematic issue of determining the best depth limit by trying all possible depth limits incrementally: First depth 0 (the initial state itself), then depth 1 (those reachable with just one step from the initial state), then depth 2, and so on. By doing this, IDS essentially combines the benefits of lightweight/memory friendly DFS and the ability of BFS that can visit neighboring states layer by layer (see Table 4.2 in Section 4.2).

Although IDS calls DLS many times, the time complexity is still $O(b^d)$ where b is the branching factor and d is the depth of the shallowest goal state. Reason: $O(b^0 + (b^0 + b^1) + (b^0 + b^1 + b^2) + ... + (b^0 + b^1 + b^2 + ... + b^d)) \leq O(c \times b^d) = O(b^d)$.

Iterative Deepening A* (IDA*)

To solve the 15-puzzle problem faster, we can use IDA* (Iterative Deepening A*) algorithm which is essentially IDS with modified DLS. IDA* calls modified DLS to try the all neighboring states in a fixed order (i.e. slide tile 0 rightwards, then upwards, then leftwards, then finally downwards—in that order; we do not use a priority queue). This modified DLS is

stopped not when it has exceeded the depth limit but when its $g(s) + h(s)$ exceeds the best known solution so far. IDA* expands the limit gradually until it hits the goal state.

The implementation of IDA* is not straightforward and we invite readers to scrutinize the given source code in the supporting website.

Source code: `ch8_01_UVa10181.cpp/java`

Exercise 8.2.5.1*: One of the hardest part in solving search problems using A* search is to find the correct admissible heuristic and to compute them efficiently as it has to be repeated many times. List down admissible heuristics that are commonly used in difficult searching problems involving A* algorithm and show how to compute them efficiently! One of them is the Manhattan distance as shown in this section.

Exercise 8.2.5.2*: Solve UVa 11212 - Editing a Book that we have discussed in depth in Section 8.2.3-8.2.4 with A* instead of bidirectional BFS! Hint: First, determine what is a suitable heuristic for this problem.

Programming Exercises solvable with More Advanced Search Techniques:

- More Challenging Backtracking Problems
    1. *UVa 00131 - The Psychic Poker Player* (backtracking with 2^5 bitmask to help deciding which card is retained in hand/exchanged with the top of deck; use 5! to shuffle the 5 cards in hand and get the best value)
    2. *UVa 00710 - The Game* (backtracking with memoization/pruning)
    3. *UVa 00711 - Dividing up* (reduce search space first before backtracking)
    4. *UVa 00989 - Su Doku* (classic Su Doku puzzle; this problem is NP complete but this instance is solvable with backtracking with pruning; use bitmask to speed up the check of available digits)
    5. *UVa 01052 - Bit Compression* (LA 3565 - WorldFinals SanAntonio06, backtracking with some form of bitmask)
    6. **UVa 10309 - Turn the Lights Off *** (brute force the first row in 2^{10}, the rest follows)
    7. *UVa 10318 - Security Panel* (the order is not important, so we can try pressing the buttons in increasing order, row by row, column by column; when pressing one button, only the 3×3 square around it is affected; therefore after we press button (i, j), light $(i-1, j-1)$ must be on (as no button afterward will affect this light); this check can be used to prune the backtracking)
    8. *UVa 10890 - Maze* (looks like a DP problem but the state—involving bitmask—cannot be memoized, fortunately the grid size is 'small')
    9. *UVa 10957 - So Doku Checker* (very similar with UVa 989; if you can solve that one, you can modify your code a bit to solve this one)
    10. **UVa 11195 - Another n-Queen Problem *** (see **Exercise 3.2.1.3*** and the discussion in Section 8.2.1)
    11. **UVa 11065 - A Gentlemen's Agreement *** (independent set, bitmask helps in speeding up the solution; see the discussion in Section 8.2.1)
    12. *UVa 11127 - Triple-Free Binary Strings* (backtracking with bitmask)
    13. *UVa 11464 - Even Parity* (brute force the first row in 2^{15}, the rest follows)
    14. *UVa 11471 - Arrange the Tiles* (reduce search space by grouping tiles of the same type; recursive backtracking)

- More Challenging State-Space Search with BFS or Dijkstra's

  1. UVa 00321 - The New Villa (s: (position, bitmask 2^{10}), print the path)
  2. *UVa 00658 - It's not a Bug ...* (s: bitmask—whether a bug is present or not, use Dijkstra's as the State-Space graph is weighted)
  3. UVa 00928 - Eternal Truths (s: (row, col, direction, step))
  4. **_UVa 00985 - Round and Round ... *_** (4 rotations is the same as 0 rotations, s: (row, col, rotation = [0..3]); find the shortest path from state [1][1][0] to state [R][C][x] where $0 \leq x \leq 3$)
  5. *UVa 01057 - Routing* (LA 3570, World Finals SanAntonio06, use Floyd Warshall's to get APSP information; then model the original problem as another weighted SSSP problem solvable with Dijkstra's)
  6. UVa 01251 - Repeated Substitution ... (LA 4637, Tokyo09, SSSP solvable with BFS)
  7. UVa 01253 - Infected Land (LA 4645, Tokyo09, SSSP solvable with BFS, tedious state modeling)
  8. UVa 10047 - The Monocyle (s: (row, col, direction, color); BFS)
  9. *UVa 10097 - The Color game* (s: (N1, N2); implicit unweighted graph; BFS)
  10. *UVa 10923 - Seven Seas* (s: (ship_position, location of enemies, location of obstacles, steps_so_far); implicit weighted graph; Dijkstra's)
  11. **_UVa 11198 - Dancing Digits *_** (s: permutation; BFS; tricky to code)
  12. **_UVa 11329 - Curious Fleas *_** (s: bitmask of 26 bits, 4 to describe the position of the die in the 4×4 grid, 16 to describe if a cell has a flea, 6 to describe the sides of the die that has a flea; use `map`; tedious to code)
  13. UVa 11513 - 9 Puzzle (reverse the role of source and destination)
  14. UVa 11974 - Switch The Lights (BFS on implicit unweighted graph)
  15. UVa 12135 - Switch Bulbs (LA 4201, Dhaka08, similar to UVa 11974)

- Meet in the Middle/A*/IDA*

  1. UVa 00652 - Eight (classical sliding block 8-puzzle problem, IDA*)
  2. **UVa 01098 - Robots on Ice *** (LA 4793, World Finals Harbin10, see the discussion in Section 8.2.2; however, there is a faster 'meet in the middle' solution for this problem)
  3. UVa 01217 - Route Planning (LA 3681, Kaohsiung06, solvable with A*/IDA*; test data likely only contains up to 15 stops which already include the starting and the last stop on the route)
  4. **UVa 10181 - 15-Puzzle Problem *** (similar as UVa 652, but this one is larger, we can use IDA*)
  5. UVa 11163 - Jaguar King (another puzzle game solvable with IDA*)
  6. **UVa 11212 - Editing a Book *** (meet in the middle, see Section 8.2.4)

- Also see some more Complete Search problems in Section 8.4

8.3 More Advanced DP Techniques

In Section 3.5, 4.7.1, 5.4, 5.6, and 6.5, we have seen the introduction of Dynamic Programming (DP) technique, several classical DP problems and their solutions, plus a gentle introduction to the easier non classical DP problems. There are several more advanced DP techniques that we have not covered in those sections. Here, we present some of them.

8.3.1 DP with Bitmask

Some of the modern DP problems require a (small) set of Boolean as one of the parameters of the DP state. This is another situation where bitmask technique can be useful (also see Section 8.2.1). This technique is suitable for DP as the integer (that represents the bitmask) can be used as the index of the DP table. We have seen this technique once when we discuss DP TSP (see Section 3.5.2). Here, we give one more example.

UVa 10911 - Forming Quiz Teams

For the abridged problem statement and the solution code of this problem, please refer to the very first problem mentioned in Chapter 1. The grandiose name of this problem is "minimum weight perfect matching on a small general weighted graph". In the general case, this problem is hard. However, if the input size is small, up to $M \leq 20$, then DP with bitmask solution can be used.

The DP with bitmask solution for this problem is simple. The matching state is represented by a `bitmask`. We illustrate this with a small example when $M = 6$. We start with a state where nothing is matched yet, i.e. `bitmask=000000`. If item 0 and item 2 are matched, we can turn on two bits (bit 0 and bit 2) at the same time via this simple bit operation, i.e. `bitmask | (1 << 0) | (1 << 2)`, thus the state becomes `bitmask=000101`. Notice that index starts from 0 and counted from the right. If from this state, item 1 and item 5 are matched next, the state will become `bitmask=100111`. The perfect matching is obtained when the state is all '1's, in this case: `bitmask=111111`.

Although there are many ways to arrive at a certain state, there are only $O(2^M)$ distinct states. For each state, we record the minimum weight of previous matchings that must be done in order to reach this state. We want a perfect matching. First, we find one 'off' bit i using one $O(M)$ loop. Then, we find the best other 'off' bit j from $[i+1 .. M-1]$ using another $O(M)$ loop and recursively match i and j. This check is again done using bit operation, i.e. we check `if (!(bitmask & (1 << i)))`—and similarly for j. This algorithm runs in $O(M \times 2^M)$. In problem UVa 10911, $M = 2N$ and $2 \leq N \leq 8$, so this DP with bitmask solution is feasible. For more details, please study the code.

> Source code: `ch8_02_UVa10911.cpp/java`

In this subsection, we have shown that DP with bitmask technique can be used to solve small instances ($M \leq 20$) of matching on general graph. In general, bitmask technique allows us to represent a small set of up to ≈ 20 items. The programming exercises in this section contain more examples when bitmask is used as *one of the parameters* of the DP state.

Exercise 8.3.1.1: Show the required DP with bitmask solution if we have to deal with "Maximum Cardinality Matching on a small general graph ($V \leq 18$)".

Exercise 8.3.1.2*: Rewrite the code `ch8_02_UVa10911.cpp/java` with the `LSOne` trick shown in Section 8.2.1 to speed it up!

8.3.2   Compilation of Common (DP) Parameters

After solving lots of DP problems (including recursive backtracking without memoization), contestants will develop a sense of which parameters are commonly selected to represent the states of the DP (or recursive backtracking) problems. Some of them are as follows:

1. Parameter: Index i in an array, e.g. $[x_0, x_1, \ldots, x_i, \ldots]$
   Transition: Extend subarray $[0..i]$ (or $[i..n\text{-}1]$), process i, take item i or not, etc
   Example: 1D Max Sum, LIS, part of 0-1 Knapsack, TSP, etc (see Section 3.5.2)

2. Parameter: Indices (i, j) in two arrays, e.g. $[x_0, x_1, \ldots, x_i] + [y_0, y_1, \ldots, y_j]$
   Transition: Extend i, j, or both, etc
   Example: String Alignment/Edit Distance, LCS, etc (see Section 6.5)

3. Parameter: Subarray (i, j) of an array $[\ldots, x_i, x_{i+1}, \ldots, x_j, \ldots]$
   Transition: Split (i, j) into $(i, k) + (k + 1, j)$ or into $(i, i + k) + (i + k + 1, j)$, etc
   Example: Matrix Chain Multiplication (see Section 9.20), etc

4. Parameter: A vertex (position) in a (usually implicit) DAG
   Transition: Process the neighbors of this vertex, etc
   Example: Shortest/Longest/Counting Paths in/on DAG, etc (Section 4.7.1)

5. Parameter: Knapsack-Style Parameter
   Transition: Decrease (or increase) current value until zero (or until threshold), etc
   Example: 0-1 Knapsack, Subset Sum, Coin Change variants, etc (see Section 3.5.2)
   Note: This parameter is not DP friendly if its range is very high.
   See tips in Section 8.3.3 if the value of this parameter can go negative.

6. Parameter: Small set (usually using bitmask technique)
   Transition: Flag one (or more) item(s) in the set to on (or off), etc
   Example: DP-TSP (see Section 3.5.2), DP with bitmask (see Section 8.3.1), etc

Note that the harder DP problems usually combine two or more parameters to represent distinct states. Try to solve more DP problems listed in this section to build your DP skills.

8.3.3   Handling Negative Parameter Values with Offset Technique

In rare cases, the possible range of a parameter used in a DP state can go negative. This causes an issue for DP solution as we map parameter value into index of a DP table. The indices of a DP table must therefore be non negative. Fortunately, this issue can be dealt easily by using offset technique to make all the indices become non negative again. We illustrate this technique with another non trivial DP problem: Free Parentheses.

UVa 1238 - Free Parentheses (ACM ICPC Jakarta08, LA 4143)

Abridged problem statement: You are given a simple arithmetic expression which consists of only *addition and subtraction* operators, i.e. `1 - 2 + 3 - 4 - 5`. You are free to put any *parentheses* to the expression anywhere and as many as you want as long as the expression is still *valid*. How many *different* numbers can you make? The answer for the simple expression above is 6:

```
1 - 2 + 3 - 4 - 5     = -7        1 - (2 + 3 - 4 - 5)   =   5
1 - (2 + 3) - 4 - 5   = -13       1 - 2 + 3 - (4 - 5)   =   3
1 - (2 + 3 - 4) - 5   = -5        1 - (2 + 3) - (4 - 5) =  -3
```

The problem specifies the following constraints: The expression consists of only $2 \leq N \leq 30$ non-negative numbers less than 100, separated by addition or subtraction operators. There is no operator before the first and after the last number.

To solve this problem, we need to make three observations:

1. We only need to put an open bracket after a '-' (negative) sign as doing so will reverse the meaning of subsequent '+' and '-' operators;

2. We can only put X close brackets if we already use X open brackets—we need to store this information to process the subproblems correctly;

3. The maximum value is $100 + 100 + ... + 100$ (100 repeated 30 times) $= 3000$ and the minimum value is $0 - 100 - ... - 100$ (one 0 followed by 29 times of negative 100) $=$ -2900—this information also need to be stored, as we will see below.

To solve this problem using DP, we need to determine which set of parameters of this problem represent distinct states. The DP parameters that are easier to identify are these two:

1. '`idx`'—the current position being processed, we need to know where we are now.

2. '`open`'—the number of open brackets so that we can produce a valid expression[4].

But these two parameters are not enough to uniquely identify the state yet. For example, this partial expression: '1-1+1-1...' has `idx = 3` (indices: 0, 1, 2, 3 have been processed), `open = 0` (cannot put close bracket anymore), which sums to 0. Then, '1-(1+1-1)...' also has the same `idx = 3`, `open = 0` and sums to 0. But '1-(1+1)-1...' has the same `idx = 3`, `open = 0`, *but* sums to -2. These two DP parameters does *not* identify unique state yet. We need one more parameter to distinguish them, i.e. the value '`val`'. This skill of identifying the correct set of parameters to represent distinct states is something that one has to develop in order to do well with DP problems. The code and its explanation are shown below:

```
void rec(int idx, int open, int val) {
  if (visited[idx][open][val+3000])    // this state has been reached before
    return;    // the +3000 trick to convert negative indices to [200..6000]
         // negative indices are not friendly for accessing a static array
  visited[idx][open][val+3000] = true;      // set this state to be reached

  if (idx == N)        // last number, current value is one of the possible
    used[val+3000] = true, return;                // result of expression

  int nval = val + num[idx] * sig[idx] * ((open % 2 == 0) ? 1 : -1);
  if (sig[idx] == -1)        // option 1: put open bracket only if sign is -
    rec(idx + 1, open + 1, nval);                  // no effect if sign is +
  if (open > 0)            // option 2: put close bracket, can only do this
    rec(idx + 1, open - 1, nval);  // if we already have some open brackets
  rec(idx + 1, open, nval);                    // option 3: normal, do nothing
}
```

[4]At `idx = N` (we have processed the last number), it is fine if we still have `open` > 0 as we can dump all the necessary closing brackets at the end of the expression, e.g.: 1 - (2 + 3 - (4 - (5))).

```
// Preprocessing: Set a Boolean array 'used' which is initially set to all
// false, then run this top-down DP by calling rec(0, 0, 0)
// The solution is the # of values in array 'used' that are flagged as true
```

As we can see from the code above, we can represent all possible states of this problem with a 3D array: `bool visited[idx][open][val]`. The purpose of this memo table `visited` is to flag if certain state has been visited or not. As 'val' ranges from -2900 to 3000 (5901 distinct values), we have to offset these range to make the range non-negative. In this example, we use a safe constant +3000. The number of states is $30 \times 30 \times 6001 \approx 5M$ with $O(1)$ processing per state. This is fast enough.

8.3.4 MLE? Consider Using Balanced BST as Memo Table

In Section 3.5.2, we have seen a DP problem: 0-1 Knapsack where the state is $(id, remW)$. Parameter id has range $[0..n\text{-}1]$ and parameter $remW$ has range $[0..S]$. If the problem author sets $n \times S$ to be quite large, it will cause the 2D array (for the DP table) of size $n \times S$ to be too large (Memory Limit Exceeded in programming contests).

Fortunately for problem like this 0-1 Knapsack, if we run the Top-Down DP on it, we will realize that not all of the states are visited (whereas the Bottom-Up DP version will have to explore all states). Therefore, we can trade runtime for smaller space by using a balanced BST (C++ STL `map` or Java `TreeMap`) as the memo table. This balanced BST will *only* record the states that are actually visited by the Top-Down DP. Thus, if there are only k visited states, we will only use $O(k)$ space instead of $n \times S$. The runtime of the Top-Down DP increases by $O(c \times \log k)$ factor. However, note that this trick is rarely useful due to the high constant factor c involved.

8.3.5 MLE/TLE? Use Better State Representation

Our 'correct' DP solution (which produces correct answer but using more computing resources) may be given Memory Limit Exceeded (MLE) or Time Limit Exceeded (TLE) verdict if the problem author used a better state representation and set larger input constraints that break our 'correct' DP solution. If that happens, we have no choice but to find a better DP state representation in order to reduce the DP table size (and subsequently speed up the overall time complexity). We illustrate this technique using an example:

UVa 1231 - ACORN (ACM ICPC Singapore07, LA 4106)
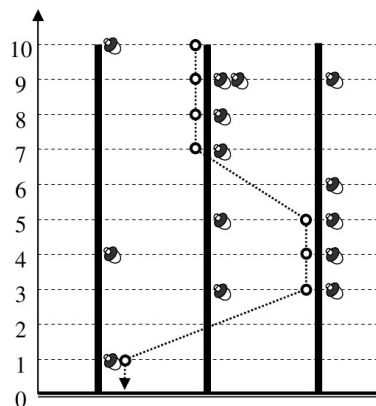
Figure 8.11: The Descent Path

Abridged problem statement: Given t oak trees, the height h of *all* trees, the height f that Jayjay the squirrel loses when it flies from one tree to another, $1 \leq t, h \leq 2000$, $1 \leq f \leq 500$, and the positions of acorns on each of the oak trees: `acorn[tree][height]`, determine the max number of acorns that Jayjay can collect in *one single descent*. Example: if $t = 3, h = 10, f = 2$ and `acorn[tree][height]` as shown in Figure 8.11, the best descent path has a total of 8 acorns (see the dotted line).

Naïve DP Solution: Use a table `total[tree][height]` that stores the best possible acorns collected when Jayjay is on a certain tree at certain height. Then Jayjay recursively tries to either go down (-1) unit on the *same* oak tree or flies (-f) unit(s) to $t - 1$ *other* oak trees from this position. On the largest test case, this requires $2000 \times 2000 = 4M$ states and $4M \times 2000 = 8B$ operations. This approach is clearly TLE.

Better DP Solution: We can actually ignore the information: "On which tree Jayjay is currently at" as just memoizing the best among them is sufficient. This is because flying to any other $t - 1$ other oak trees decreases Jayjay's height in the same manner. Set a table: `dp[height]` that stores the best possible acorns collected when Jayjay is at this `height`. The bottom-up DP code that requires only $2000 = 2K$ states and time complexity of $2000 \times 2000 = 4M$ is shown below:

```
for (int tree = 0; tree < t; tree++)                         // initialization
  dp[h] = max(dp[h], acorn[tree][h]);

for (int height = h - 1; height >= 0; height--)
  for (int tree = 0; tree < t; tree++) {
    acorn[tree][height] +=
      max(acorn[tree][height + 1],              // from this tree, +1 above
      ((height + f <= h) ? dp[height + f] : 0)); // from tree at height + f
    dp[height] = max(dp[height], acorn[tree][height]);   // update this too
  }

printf("%d\n", dp[0]);                          // the solution is stored here
```

Source code: `ch8_03_UVa1231.cpp/java`

When the size of naïve DP states are too large that causes the overall DP time complexity to be not-doable, think of another more efficient (but usually not obvious) way to represent the possible states. Using a good state representation is a potential major speed up for a DP solution. Remember that no programming contest problem is unsolvable, the problem author must have known a trick.

8.3.6 MLE/TLE? Drop One Parameter, Recover It from Others

Another known trick to reduce the memory usage of a DP solution (and thereby speed up the solution) is to drop one important parameter which can be recovered by using the other parameter(s). We use one ACM ICPC World Finals problem to illustrate this technique.

UVa 1099 - Sharing Chocolate (ACM ICPC World Finals Harbin10, LA 4794)

Abridged problem description: Given a big chocolate bar of size $1 \leq w, h \leq 100$, $1 \leq n \leq 15$ friends, and the size request of each friend. Can we break the chocolate by using horizontal and vertical cuts so that each friend gets *one piece* of chocolate bar of his chosen size?

For example, see Figure 8.12 (left). The size of the original chocolate bar is $w = 4$ and $h = 3$. If there are 4 friends, each requesting a chocolate piece of size $\{6, 3, 2, 1\}$, respectively, then we can break the chocolate into 4 parts using 3 cuts as shown in Figure 8.12 (right).
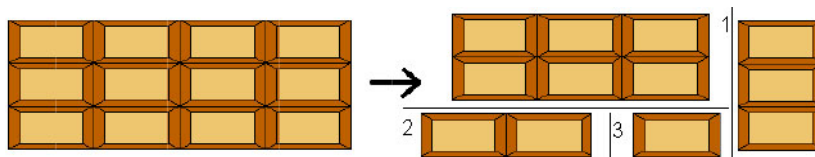
Figure 8.12: Illustration for ACM ICPC WF2010 - J - Sharing Chocolate

For contestants who are already familiar with DP technique, then the following ideas should easily come to mind: First, if sum of all requests is not the same as $w \times h$, then there is no solution. Otherwise, we can represent a distinct state of this problem using three parameters: $(w, h, bitmask)$ where w and h are the current dimension of the chocolate that we are currently considering; and $bitmask$ is the subset of friends that already have chocolate piece of their chosen size. However, a quick analysis shows that this requires a DP table of size $100 \times 100 \times 2^{15} = 327M$. This is too much for programming contest.

A better state representation is to use only two parameters, either: $(w, bitmask)$ or $(h, bitmask)$. Without loss of generality, we adopt $(w, bitmask)$ formulation. With this formulation, we can 'recover' the required value h via `sum(bitmask) / w`, where `sum(bitmask)` is the sum of the piece sizes requested by satisfied friends in $bitmask$ (i.e. all the 'on' bits of $bitmask$). This way, we have all the required parameters: w, h, and $bitmask$, but we only use a DP table of size $100 \times 2^{15} = 3M$. This one is doable.

Base cases: If $bitmask$ only contains 1 'on' bit and the requested chocolate size of that person equals to $w \times h$, we have a solution. Otherwise we do not have a solution.

For general cases: If we have a chocolate piece of size $w \times h$ and a current set of satisfied friends $bitmask = bitmask_1 \bigcup bitmask_2$, we can do either horizontal or vertical cut so that one piece is to serve friends in $bitmask_1$ and the other is to serve friends in $bitmask_2$.

The worst case time complexity for this problem is still huge, but with proper pruning, this solution runs within time limit.

Exercise 8.3.6.1*: Solve UVa 10482 - The Candyman Can and UVa 10626 - Buying Coke that use this technique. Determine which parameter is the most effective to be dropped but can still be recovered from other parameters.

Other than several DP problems in Section 8.4, there are a few more DP problems in Chapter 9 which are not listed in this Chapter 8 as they are considered *rare*. They are:

1. Section 9.2: Bitonic Traveling Salesman Problem (we also re-highlight the 'drop one parameter and recover it from others' technique),

2. Section 9.5: Chinese Postman Problem (another usage of DP with bitmask to solve the minimum weight perfect matching on small general weighted graph),

3. Section 9.20: Matrix Chain Multiplication (a classic DP problem),

4. Section 9.21: Matrix Power (we can speed up the DP transitions for *some* rare DP problems from $O(n)$ to $O(\log n)$ by rewriting the DP recurrences as matrix multiplication),

5. Section 9.22: Max Weighted Independent Set (on tree) can be solved with DP,

6. Section 9.33: Sparse Table Data Structure uses DP.

Programming Exercises related to More Advanced DP:

- DP level 2 (slightly harder than those listed in Chapter 3, 4, 5, and 6)

  1. **UVa 01172 - The Bridges of ... *** (LA 3986, DP non classic, a bit of matching flavor but with left to right and OS type constraints)
  2. **UVa 01211 - Atomic Car Race *** (LA 3404, Tokyo05, precompute array T[L], the time to run a path of length L; DP with one parameter i, where i is the checkpoint where we change tire; if i = n, we do not change the tire)
  3. UVa 10069 - Distinct Subsequences (use Java BigInteger)
  4. UVa 10081 - Tight Words (use doubles)
  5. UVa 10364 - Square (bitmask technique can be used)
  6. *UVa 10419 - Sum-up the Primes* (print path, prime)
  7. *UVa 10536 - Game of Euler* (model the 4×4 board and 48 possible pins as bitmask; then this is a simple two player game; also see Section 5.8)
  8. UVa 10651 - Pebble Solitaire (small problem size; doable with backtracking)
  9. *UVa 10690 - Expression Again* (DP Subset Sum, with negative offset technique, with addition of simple math)
  10. UVa 10898 - Combo Deal (similar to DP + bitmask; store state as integer)
  11. **UVa 10911 - Forming Quiz Teams *** (elaborated in this section)
  12. *UVa 11088 - End up with More Teams* (similar to UVa 10911, but this time it is about matching of *three* persons to one team)
  13. UVa 11832 - Account Book (interesting DP; s: (id, val); use offset to handle negative numbers; t: plus or minus; print solution)
  14. UVa 11218 - KTV (still solvable with complete search)
  15. *UVa 12324 - Philip J. Fry Problem* (must make an observation that sphere $> n$ is useless)

- DP level 3

  1. UVa 00607 - Scheduling Lectures (returns pair of information)
  2. *UVa 00702 - The Vindictive Coach* (the implicit DAG is not trivial)
  3. *UVa 00812 - Trade on Verweggistan* (mix between greedy and DP)
  4. UVa 00882 - The Mailbox ... (s: (lo, hi, mailbox_left); try all)
  5. **UVa 01231 - ACORN *** (LA 4106, Singapore07, DP with dimension reduction, discussed in this section)
  6. **UVa 01238 - Free Parentheses *** (LA 4143, Jakarta08, problem author: Felix Halim, discussed in this section)
  7. UVa 01240 - ICPC Team Strategy (LA 4146, Jakarta08)
  8. UVa 01244 - Palindromic paths (LA 4336, Amritapuri08, store the best path between i, j; the DP table contains strings)
  9. *UVa 10029 - Edit Step Ladders* (use `map` as memo table)
  10. *UVa 10032 - Tug of War* (DP Knapsack with optimization to avoid TLE)
  11. *UVa 10154 - Weights and Measures* (LIS variant)
  12. UVa 10163 - Storage Keepers (try all possible safe line L and run DP; s: id, N_left; t: hire/skip person 'id' for looking at K storage)
  13. UVa 10164 - Number Game (a bit number theory (modulo), backtracking; do memoization on DP state: (sum, taken))

14. UVa 10271 - Chopsticks (Observation: The 3rd chopstick can be any chopstick, we must greedily select adjacent chopstick, DP state: (pos, k_left), transition: Ignore this chopstick, or take this chopstick and the chopstick immediately next to it, then move to pos + 2; prune infeasible states when there are not enough chopsticks left to form triplets.)

15. UVa 10304 - Optimal Binary ... (classical DP, requires 1D range sum and Knuth-Yao speed up to get $O(n^2)$ solution)

16. *UVa 10604 - Chemical Reaction* (the mixing can be done with any pair of chemicals until there are only two chemicals left; memoize the remaining chemicals with help of `map`; sorting the remaining chemicals help increasing the number of hits to the memo table)

17. *UVa 10645 - Menu* (s: (days_left, budget_left, prev_dish, prev_dish_count); the first two parameters are knapsack-style parameter; the last two parameters are used to determine the price of that dish as first, second, and subsequent usage of the dish has different values)

18. UVa 10817 - Headmaster's Headache (s: (id, bitmask); space: $100 \times 2^{2*8}$)

19. *UVa 11002 - Towards Zero* (a simple DP; use negative offset technique)

20. *UVa 11084 - Anagram Division* (using `next_permutation`/brute force is probably not the best approach, there is a DP formulation for this)

21. UVa 11285 - Exchange Rates (maintain the best CAD & USD each day)

22. **UVa 11391 - Blobs in the Board *** (DP with bitmask on 2D grid)

23. *UVa 12030 - Help the Winners* (s: (idx, bitmask, all1, has2); t: try all shoes that has not been matched to the girl that choose dress 'idx')

- DP level 4

    1. UVa 00473 - Raucous Rockers (the input constraint is not clear; therefore use resizeable `vector` and compact states)

    2. **UVa 01099 - Sharing Chocolate *** (LA 4794, World Finals Harbin10, discussed in this section)

    3. **UVa 01220 - Party at Hali-Bula *** (LA 3794, Tehran06; Maximum Independent Set (MIS) problem on tree; DP; also check if the MIS is unique)

    4. UVa 01222 - Bribing FIPA (LA 3797, Tehran06, DP on Tree)

    5. **UVa 01252 - Twenty Questions *** (LA 4643, Tokyo09, DP, s: (bitmask1, bitmask2) where bitmask1 describes the features that we decide to ask and bitmask2 describes the answers of the features that we ask)

    6. *UVa 10149 - Yahtzee* (DP with bitmask; uses card rules; tedious)

    7. UVa 10482 - The Candyman Can (see **Exercise 8.3.6.1***)

    8. UVa 10626 - Buying Coke (see **Exercise 8.3.6.1***)

    9. *UVa 10722 - Super Lucky Numbers* (needs Java BigInteger; DP formulation must be efficient to avoid TLE; state: (N_digits_left, B, first, previous_digit_is_one) and use a bit of simple combinatorics to get the answer)

    10. *UVa 11125 - Arrange Some Marbles* (counting paths in implicit DAG; 8 dimensional DP)

    11. *UVa 11133 - Eigensequence* (the implicit DAG is not trivial)

    12. *UVa 11432 - Busy Programmer* (the implicit DAG is not trivial)

    13. UVa 11472 - Beautiful Numbers (DP state with four parameters)

- Also see some more DP problems in Section 8.4 and in Chapter 9

8.4 Problem Decomposition

While there are only 'a few' basic data structures and algorithms tested in programming contest problems (we believe that many of them have been covered in this book), the harder problems may require a *combination* of two (or more) algorithms and/or data structures. To solve such problems, we must first decompose the components of the problems so that we can solve each component independently. To be able to do so, we must first be familiar with the individual components (the content of Chapter 1 up to Section 8.3).

Although there are $_NC_2$ possible combinations of two out of N algorithms and/or data structures, not all of the combinations make sense. In this section, we compile and list down some[5] of the *more common* combinations of two algorithms and/or data structures based on our experience in solving ≈ 1675 UVa online judge problems. We end this section with the discussion of the rare combination of *three* algorithms and/or data structures.

8.4.1 Two Components: Binary Search the Answer and Other

In Section 3.3.1, we have seen binary search the answer on a (simple) simulation problem that does not depend on the fancier algorithms listed after Section 3.3.1. Actually, this technique can be combined with some other algorithms in Section 3.4 - Section 8.3. Several variants that we have encountered so far are binary search the answer plus:

- Greedy algorithm (discussed in Section 3.4), e.g. UVa 714, 11516,

- Graph connectivity test (discussed in Section 4.2), e.g. UVa 295, 10876,

- SSSP algorithm (discussed in Section 4.4), e.g. UVa 10816, IOI 2009 (Mecho),

- Max Flow algorithm (discussed in Section 4.6), e.g. UVa 10983,

- MCBM algorithm (discussed in Section 4.7.4), e.g. UVa 1221, 10804, 11262,

- BigInteger operations (discussed in Section 5.3), e.g. UVa 10606,

- Geometry formulas (discussed in Section 7.2), e.g. UVa 1280, 10566, 10668, 11646.

In this section, we write two more examples of using binary search the answer technique. This combination of binary search the answer plus another algorithm can be spotted by asking this question: "If we guess the required answer (in binary search fashion) and assume this answer is true, will the original problem be solvable or not (a True/False question)?".

Binary Search the Answer plus Greedy algorithm

Abridged problem description of UVa 714 - Copying Books: You are given $m \leq 500$ books numbered $1, 2, \ldots, m$ that may have different number of pages $(p_1, p_2, \ldots, p_m)$. You want to make one copy of each of them. Your task is to assign these books among k scribes, $k \leq m$. Each book can be assigned to a single scriber only, and every scriber must get a *continuous sequence* of books. That means, there exists an increasing succession of numbers $0 = b_0 < b_1 < b_2, \cdots < b_{k-1} \leq b_k = m$ such that i-th scriber $(i > 0)$ gets a sequence of books with numbers between $b_{i-1} + 1$ and b_i. Each scribe copies pages at the same rate. Thus, the time needed to make one copy of each book is determined by the scriber who is assigned the most work. Now, you want to determine: "What is the minimum number of pages copied by the scriber with the most work?".

---

[5]This list is not and probably will not be exhaustive.

There exists a Dynamic Programming solution for this problem, but this problem can also be solved by guessing the answer in binary search fashion. We will illustrate this with an example when $m = 9$, $k = 3$ and $p_1, p_2, \ldots, p_9$ are 100, 200, 300, 400, 500, 600, 700, 800, and 900, respectively.

If we guess that the *answer* = 1000, then the problem becomes 'simpler', i.e. If the scriber with the most work can only copy up to 1000 pages, can this problem be solved? The answer is 'no'. We can greedily assign the jobs from book 1 to book m as follows: {100, 200, 300, 400} for scribe 1, {500} for scribe 2, {600} for scribe 3. But if we do this, we still have 3 books {700, 800, 900} unassigned. Therefore the answer must be > 1000.

If we guess *answer* = 2000, then we can greedily assign the jobs as follows: {100, 200, 300, 400, 500} for scribe 1, {600, 700} for scribe 2, and {800, 900} for scribe 3. All books are copied and we still have some slacks, i.e. scribe 1, 2, and 3 still have {500, 700, 300} unused potential. Therefore the answer must be ≤ 2000.

This *answer* is binary-searchable between $[lo..hi]$ where $lo = max(p_i), \forall i \in [1..m]$ (the number of pages of the thickest book) and $hi = p_1 + p_2 + \ldots + p_m$ (the sum of all pages from all books). And for those who are curious, the optimal *answer* for the test case in this example is 1700. The time complexity of this solution is $O(m \log hi)$. Notice that this extra log factor is usually negligible in programming contest environment.

Binary Search the Answer plus Geometry formulas

We use UVa 11646 - Athletics Track for another illustration of Binary Search the Answer tecnique. The abridged problem description is as follows: Examine a rectangular soccer field with an athletics track as seen in Figure 8.13—left where the two arcs on both sides (arc1 and arc2) are from the same circle centered in the middle of the soccer field. We want the length of the athletics track (L1 + arc1 + L2 + arc2) to be exactly 400m. If we are given the ratio of the length L and width W of the soccer field to be $a : b$, what should be the actual length L and width W of the soccer field that satisfy the constraints above?
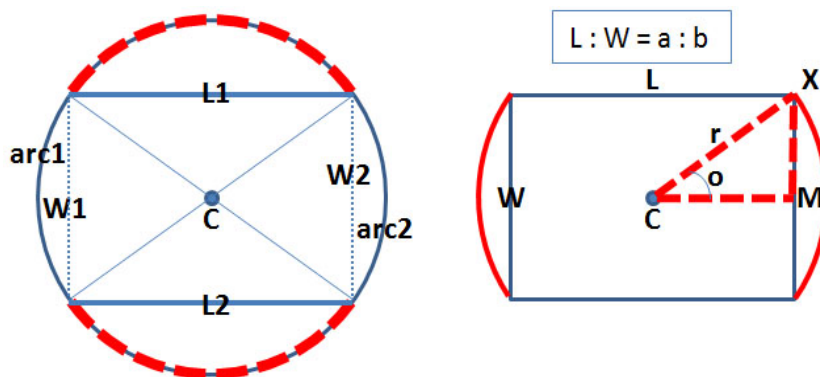
Figure 8.13: Athletics Track (from UVa 11646)

It is quite hard (but not impossible) to obtain the solution with pen and paper strategy (analytical solution), but with the help of a computer and binary search the answer (actually bisection method) technique, we can find the solution easily.

We binary search the value of L. From L, we can get $W = b/a \times L$. The expected length of an arc is $(400 - 2 \times L)/2$. Now we can use Trigonometry to compute the radius r and the angle o via triangle CMX (see Figure 8.13—right). $CM = 0.5 \times L$ and $MX = 0.5 \times W$. With r and o, we can compute the actual arc length. We then compare this value with the expected arc length to decide whether we have to increase or decrease the length L. The snippet of the code is shown below.

```
lo = 0.0; hi = 400.0;              // this is the possible range of the answer
while (fabs(lo - hi) > 1e-9) {
  L = (lo + hi) / 2.0;                              // do bisection method on L
  W = b / a * L;                      // W can be derived from L and ratio a : b
  expected_arc = (400 - 2.0 * L) / 2.0;                      // reference value

  CM = 0.5 * L; MX = 0.5 * W;                        // apply Trigonometry here
  r = sqrt(CM * CM + MX + MX);
  angle = 2.0 * atan(MX / CM) * 180.0 / PI;     // arc's angle = 2x angle o
  this_arc = angle / 360.0 * PI * (2.0 * r);      // compute the arc value

  if (this_arc > expected_arc) hi = L; else lo = L;  // decrease/increase L
}
printf("Case %d: %.12lf %.12lf\n", caseNo++, L, W);
```

Exercise 8.4.1.1*: Prove that other strategies will not be better than the greedy strategy mentioned for the UVa 714 solution above?

Exercise 8.4.1.2*: Derive analytical solution for UVa 11646 instead of using this binary search the answer technique.

8.4.2 Two Components: Involving 1D Static RSQ/RMQ

This combination should be rather easy to spot. The problem involves *another* algorithm to populate the content of a *static* 1D array (that will not be changed anymore once it is populated) and then there will be *many* Range Sum/Minimum/Maximum Queries (RSQ/RMQ) on this static 1D array. Most of the time, these RSQs/RMQs are asked at the output phase of the problem. But sometimes, these RSQs/RMQs are used to speed up the internal mechanism of the other algorithm to solve the problem.

The solution for 1D Static RSQ with Dynamic Programming has been discussed in Section 3.5.2. For 1D Static RMQ, we have the Sparse Table Data Structure (which is a DP solution) that is discussed in Section 9.33. Without this RSQ/RMQ DP speedup, the other algorithm that is needed to solve the problem usually ends up receiving the TLE verdict.

As a simple example, consider a simple problem that asks how many primes there are in various query ranges $[a..b]$ ($2 \le a \le b \le 1000000$). This problem clearly involves Prime Number generation (e.g. Sieve algorithm, see Section 5.5.1). But since this problem has $2 \le a \le b \le 1000000$, we will get TLE if we keep answering each query in $O(b - a + 1)$ time by iterating from a to b, especially if the problem author purposely set $b - a + 1$ to be near 1000000 at (almost) every query. We need to speed up the output phase into $O(1)$ per query using 1D Static RSQ DP solution.

8.4.3 Two Components: Graph Preprocessing and DP

In this subsection, we want to highlight a problem where graph pre-processing is one of the components as the problem clearly involves some graphs and DP is the other component. We show this combination with two examples.

SSSP/APSP plus DP TSP

We use UVa 10937 - Blackbeard the Pirate to illustrate this combination of SSSP/APSP plus DP TSP. The SSSP/APSP is usually used to transform the input (usually an implicit graph/grid) into another (usually smaller) graph. Then we run Dynamic Programming solution for TSP on the second (usually smaller) graph.

The given input for this problem is shown on the left of the diagram below. This is a 'map' of an island. Blackbeard has just landed at this island and at position labeled with a '@'. He has stashed up to 10 treasures in this island. The treasures are labeled with exclamation marks '!'. There are angry natives labeled with '*'. Blackbeard has to stay away at least 1 square away from the angry natives in any of the eight directions. Blackbeard wants to grab all his treasures and go back to his ship. He can only walk on land '.' cells and not on water '~' cells nor on obstacle cells '#'.

```
        Input:           Index @ and !      The APSP Distance Matrix
   Implicit Graph     Enlarge * with X      A complete (small) graph
     ~~~~~~~~~~           ~~~~~~~~~~         --------------------
     ~~!!!###~~           ~~123###~~          | 0| 1| 2| 3| 4| 5|
     ~##...###~           ~##..X###~          --------------------
     ~#....*##~           ~#..XX*##~         |0| 0|11|10|11| 8| 8|
     ~#!..**~~~           ~#4.X**~~~         |1|11| 0| 1| 2| 5| 9|
     ~~....~~~~   ==>     ~~..XX~~~~   ==>   |2|10| 1| 0| 1| 4| 8|
     ~~~....~~~           ~~~....~~~         |3|11| 2| 1| 0| 5| 9|
     ~~..~..@~~           ~~..~..0~~         |4| 8| 5| 4| 5| 0| 6|
     ~#!.~~~~~~           ~#5.~~~~~~         |5| 8| 9| 8| 9| 6| 0|
     ~~~~~~~~~~           ~~~~~~~~~~         --------------------
```

This is clearly a TSP problem (see Section 3.5.3), but before we can use DP TSP solution, we have to first transform the input into a distance matrix.

In this problem, we are only interested in the '@' and the '!'s. We give index 0 to '@' and give positive indices to the other '!'s. We enlarge the reach of each '*' by replacing the '.' around the '*' with a 'X'. Then we run BFS on this unweighted implicit graph starting from '@' and all the '!', by only stepping on cells labeled with '.' (land cells). This gives us the All-Pairs Shortest Paths (APSP) distance matrix as shown in the diagram above.

Now, after having the APSP distance matrix, we can run DP TSP as shown in Section 3.5.3 to obtain the answer. In the test case shown above, the optimal TSP tour is: 0-5-4-1-2-3-0 with cost = 8+6+5+1+1+11 = 32.

SCC Contraction plus DP Algorithm on DAG

In some modern problems involving *directed* graph, we have to deal with the Strongly Connected Components (SCCs) of the directed graph (see Section 4.2.9). One of the newer variants is the problem that requires all SCCs of the given directed graph to be *contracted* first to form larger vertices (which we name as super vertices). The original directed graph is not guaranteed to be acyclic, thus we cannot immediately apply DP techniques on such graph. But when the SCCs of a directed graph are contracted, the resulting graph of super vertices is a DAG (see Figure 4.9 for an example). If you recall our discussion in Section 4.7.1, DAG is very suitable for DP techniques as it is acyclic.

UVa 11324 - The Largest Clique is one such problem. This problem in short, is about finding the longest path on the DAG of contracted SCCs. Each super vertex has weight that represents the number of original vertices that are contracted into that super vertex.

8.4.4 Two Components: Involving Graph

This type of problem combinations can be spotted as follows: One clear component is a graph algorithm. However, we need another supporting algorithm, which is usually some sort of mathematics or geometric rule (to build the underlying graph) or even another supporting graph algorithm. In this subsection, we illustrate one such example.

In Section 2.4.1, we have mentioned that for some problems, the underlying graph does not need to be stored in any graph specific data structures (implicit graph). This is possible if we can derive the edges of the graph easily or via some rules. UVa 11730 - Number Transformation is one such problem.

While the problem description is all mathematics, the main problem is actually a Single-Source Shortest Paths (SSSP) problem on unweighted graph solvable with BFS. The underlying graph is generated on the fly during the execution of the BFS. The source is the number S. Then, every time BFS process a vertex u, it enqueues unvisited vertex $u + x$ where x is a prime factor of u that is not 1 or u itself. The BFS layer count when target vertex T is reached is the minimum number of transformations needed to transform S into T according to the problem rules.

8.4.5 Two Components: Involving Mathematics

In this problem combination, one of the components is clearly a mathematics problem, but it is not the only one. It is usually not graph as otherwise it will be classified in the previous subsection above. The other component is usually recursive backtracking or binary search. It is also possible to have two different mathematics algorithms in the same problem. In this subsection, we illustrate one such example.

UVa 10637 - Coprimes is the problem of partitioning S ($0 < S \le 100$) into t ($0 < t \le 30$) co-prime numbers. For example, for $S = 8$ and $t = 3$, we can have $1 + 1 + 6$, $1 + 2 + 5$, or $1 + 3 + 4$. After reading the problem description, we will have a strong feeling that this is a mathematics (number theory) problem. However, we will need more than just Sieve of Eratosthenes algorithm to generate the primes and GCD algorithm to check if two numbers are co-prime, but also a recursive backtracking routine to generate all possible partitions.

8.4.6 Two Components: Complete Search and Geometry

Many (computational) geometry problems are brute-force-able (although some requires Divide and Conquer-based solution). When the given input constraints allow for such Complete Search solution, do not hesitate to go for it.

For example, UVa 11227 - The silver bullet boils down into this problem: Given N ($1 \le N \le 100$) points on a 2D plane, determine the maximum number of points that are collinear. We can afford to use the following $O(N^3)$ Complete Search solution as $N \le 100$ (there is a better solution). For each pair of point i and j, we check the other N-2 points if they are collinear with line $i - j$. This solution can be easily written with three nested loops and the `bool collinear(point p, point q, point r)` function shown in Section 7.2.2.

8.4.7 Two Components: Involving Efficient Data Structure

This problem combination usually appear in some 'standard' problem but with *large* input constraint such that we have to use a more efficient data structure to avoid TLE.

For example, UVa 11967-Hic-Hac-Hoe is an extension of a board game Tic-Tac-Toe. Instead of the small 3×3 board, this time the board size is 'infinite'. Thus, there is no way we can record the board using a 2D array. Fortunately, we can store the coordinates of the 'noughts' and 'crosses' in a balanced BST and refer to this BST to check the game state.

8.4.8 Three Components

In Section 8.4.1-8.4.7, we have seen various examples of problems involving two components. In this subsection, we show two examples of rare combinations of three different algorithms and/or data structures.

Prime Factors, DP, Binary Search

UVa 10856 - Recover Factorial can be abridged as follow: "Given N, the number of prime factors in $X!$, what is the minimum possible value of X? ($N \leq 10000001$)". This problem is quite challenging. To make it doable, we have to decompose it into several components.

First, we compute the number of prime factors of an integer i and store it in a table `NumPF[i]` with the following recurrence: If i is a prime, then `NumPF[i] = 1` prime factor; else if $i = PF \times i'$, then `NumPF[i] = 1 +` the number of prime factors of i'. We compute this number of prime factors $\forall i \in$ `[1..2703665]`. The upper bound of this range is obtained by trial and error according to the limits given in the problem description.

Then, the second part of the solution is to *accumulate* the number of prime factors of $N!$ by setting `NumPF[i] += NumPF[i-1];` $\forall i \in$ `[1..N]`. Thus, `NumPF[N]` contains the number of prime factors of $N!$. This is the DP solution for the 1D Static RSQ problem.

Now, the third part of the solution should be obvious: We can do binary search to find the index X such that `NumPF[X] = N`. If there is no answer, we output "Not possible.".

Complete Search, Binary Search, Greedy

In this write up, we discuss an ACM ICPC World Finals programming problem that combines *three* problem solving paradigms that we have learned in Chapter 3, namely: Complete Search, Divide & Conquer (Binary Search), and Greedy.

ACM ICPC World Finals 2009 - Problem A - A Careful Approach, LA 4445

Abridged problem description: You are given a scenario of airplane landings. There are $2 \leq n \leq 8$ airplanes in the scenario. Each airplane has a time window during which it can safely land. This time window is specified by two integers a_i and b_i, which gives the beginning and end of a closed interval $[a_i..b_i]$ during which the i-th plane can land safely. The numbers a_i and b_i are specified in minutes and satisfy $0 \leq a_i \leq b_i \leq 1440$ (24 hours). In this problem, you can assume that the plane landing time is negligible. Your tasks are:

1. Compute an **order for landing all airplanes** that respects these time windows.
   HINT: order = permutation = Complete Search?

2. Furthermore, the airplane landings should be stretched out **as much as possible** so that the minimum achievable time gap between successive landings is as large as possible. For example, if three airplanes land at 10:00am, 10:05am, and 10:15am, then the smallest gap is five minutes, which occurs between the first two airplanes. Not all gaps have to be the same, but the smallest gap should be as large as possible.
   HINT: Is this similar to 'interval covering' problem (see Section 3.4.1)?

3. Print the answer split into minutes and seconds, rounded to the closest second.

See Figure 8.14 for illustration:
line = the safe landing time window of a plane.
star = the plane's optimal landing schedule.
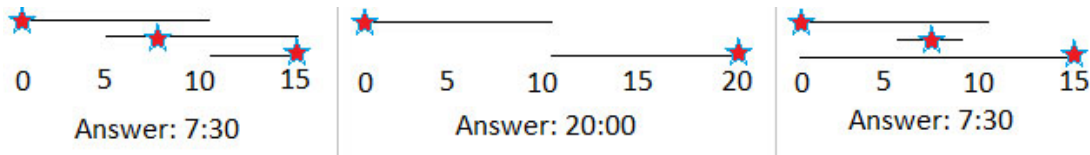
Figure 8.14: Illustration for ACM ICPC WF2009 - A - A Careful Approach

Solution: Since the number of planes is at most 8, an optimal solution can be found by simply trying all $8! = 40320$ possible orders for the planes to land. This is the **Complete Search** component of the problem which can be easily implemented using `next_permutation` in C++ STL `algorithm`.

Now, for each specific landing order, we want to know the largest possible landing window. Suppose we guess that the answer is a certain window length L. We can greedily check whether this L is feasible by forcing the first plane to land as soon as possible and the subsequent planes to land in `max(a[that plane], previous landing time + ` L`)`. This is the **Greedy** component.

A window length L that is too long/short will cause `lastLanding` (see the code below) to overshoot/undershoot `b[last plane]`, so we have to decrease/increase L. We can binary search the answer L. This is the **Divide and Conquer** component of this problem. As we only want the answer rounded to the nearest integer, stopping binary search when the error $\epsilon < $ 1e-3 is enough. For more details, please study our source code shown below.

```cpp
// World Finals Stockholm 2009, A - A Careful Approach, UVa 1079, LA 4445

#include <algorithm>
#include <cmath>
#include <cstdio>
using namespace std;

int i, n, caseNo = 1, order[8];
double a[8], b[8], L, maxL;

double greedyLanding() {  // with certain landing order, and certain L, try
        // landing those planes and see what is the gap to b[order[n - 1]]
  double lastLanding = a[order[0]];      // greedy, 1st aircraft lands ASAP
  for (i = 1; i < n; i++) {                          // for the other aircrafts
    double targetLandingTime = lastLanding + L;
    if (targetLandingTime <= b[order[i]])
       // can land: greedily choose max of a[order[i]] or targetLandingTime
      lastLanding = max(a[order[i]], targetLandingTime);
    else
      return 1;
  }
  // return +ve value to force binary search to reduce L
  // return -ve value to force binary search to increase L
  return lastLanding - b[order[n - 1]];
}
```

```
int main() {
  while (scanf("%d", &n), n) {                              // 2 <= n <= 8
    for (i = 0; i < n; i++) {    // plane i land safely at interval [ai, bi]
      scanf("%lf %lf", &a[i], &b[i]);
      a[i] *= 60; b[i] *= 60;  // originally in minutes, convert to seconds
      order[i] = i;
    }

    maxL = -1.0;                                 // variable to be searched for
    do {                              // permute plane landing order, up to 8!
      double lo = 0, hi = 86400;               // min 0s, max 1 day = 86400s
      L = -1;                               // start with an infeasible solution
      while (fabs(lo - hi) >= 1e-3) {         // binary search L, EPS = 1e-3
        L = (lo + hi) / 2.0;    // we want the answer rounded to nearest int
        double retVal = greedyLanding();                 // round down first
        if (retVal <= 1e-2) lo = L;                       // must increase L
        else                hi = L;          // infeasible, must decrease L
      }
      maxL = max(maxL, L);              // get the max over all permutations
    }
    while (next_permutation(order, order + n));     // try all permutations

    // other way for rounding is to use printf format string: %.0lf:%0.2lf
    maxL = (int)(maxL + 0.5);                         // round to nearest second
    printf("Case %d: %d:%0.2d\n", caseNo++, (int)(maxL/60), (int)maxL%60);
  }

  return 0;
}
```

Source code: `ch8_04_UVa1079.cpp/java`

Exercise 8.4.8.1: The given code above is Accepted, but it uses 'double' data type for `lo`, `hi`, and `L`. This is actually unnecessary as all computations can be done in integers. Also, instead of using `while (fabs(lo - hi) >= 1e-3)`, use `for (int i = 0; i < 50; i++)` instead! Please rewrite this code!

Programming Exercises related to Problem Decomposition:

- Two Components - Binary Search the Answer and Other

  1. UVa 00714 - Copying Books (binary search the answer + greedy)
  2. UVa 01221 - Against Mammoths (LA 3795, Tehran06, binary search the answer + MCBM (perfect matching); use the augmenting path algorithm to compute MCBM—see Section 4.7.4)
  3. *UVa 01280 - Curvy Little Bottles* (LA 6027, World Finals Warsaw12, binary search the answer and geometric formula)
  4. *UVa 10372 - Leaps Tall Buildings ...* (binary search the answer + Physics)
  5. UVa 10566 - Crossed Ladders (bisection method)
  6. *UVa 10606 - Opening Doors* (the solution is simply the highest square number $\leq N$, but this problem involves BigInteger; we use a (rather slow) binary search the answer technique to obtain $\sqrt{N}$)
  7. UVa 10668 - Expanding Rods (bisection method)
  8. UVa 10804 - Gopher Strategy (similar to UVa 11262)
  9. UVa 10816 - Travel in Desert (binary search the answer + Dijkstra's)
  10. **UVa 10983 - Buy one, get ... *** (binary search the answer + max flow)
  11. **UVa 11262 - Weird Fence *** (binary search the answer + MCBM)
  12. **UVa 11516 - WiFi *** (binary search the answer + greedy)
  13. UVa 11646 - Athletics Track (the circle is at the center of track)
  14. *UVa 12428 - Enemy at the Gates* (binary search the answer + a bit of graph theory about bridges as outlined in Chapter 4)
  15. IOI 2009 - Mecho (binary search the answer + BFS)

- Two Components - Involving DP 1D RSQ/RMQ

  1. UVa 00967 - Circular (similar to UVa 897, but this time the output part can be speed up using DP 1D range sum)
  2. UVa 10200 - Prime Time (complete search, test if isPrime($n^2 + n + 41$) $\forall n \in$ [a..b]; FYI, this prime generating formula $n^2 + n + 41$ was found by Leonhard Euler; for $0 \leq n \leq 40$, it works; however, it does not have good accuracy for larger n; finally use DP 1D RSQ to speed up the solution)
  3. UVa 10533 - Digit Primes (sieve; check if a prime is a digit prime; DP 1D range sum)
  4. UVa 10871 - Primed Subsequence (need 1D Range Sum Query)
  5. **UVa 10891 - Game of Sum *** (Double DP; The first DP is the standard 1D Range Sum Query between two indices: i, j. The second DP evaluates the Decision Tree with state (i, j) and try all splitting points; minimax.)
  6. ***UVa 11105 - Semi-prime H-numbers **** (need 1D Range Sum Query)
  7. **UVa 11408 - Count DePrimes *** (need 1D Range Sum Query)
  8. *UVa 11491 - Erasing and Winning* (greedy, optimized with Sparse Table data structure to deal with the static RMQ)
  9. *UVa 12028 - A Gift from ...* (generate the array; sort it; prepare 1D Range Sum Query; then the solution will be much simpler)

- Two Components - Graph Preprocessing and DP

  1. ***UVa 00976 - Bridge Building **** (use a kind of flood fill to separate north and south banks; use it to compute the cost of installing a bridge at each column; a DP solution should be quite obvious after this preprocessing)
  2. UVa 10917 - A Walk Through the Forest (counting paths in DAG; but first, you have to build the DAG by running Dijkstra's algorithm from 'home')
  3. UVa 10937 - Blackbeard the Pirate (BFS → TSP, then DP or backtracking; discussed in this section)
  4. UVa 10944 - Nuts for nuts.. (BFS → TSP, then use DP, $n \leq 16$)
  5. **UVa 11324 - The Largest Clique *** (longest paths on DAG; but first, you have to transform the graph into DAG of its SCCs; toposort)
  6. **UVa 11405 - Can U Win? *** (BFS from 'k' & each 'P'—max 9 items; then use DP-TSP)
  7. *UVa 11693 - Speedy Escape* (compute shotest paths information using Floyd Warshall's; then use DP)
  8. UVa 11813 - Shopping (Dijsktra's → TSP, then use DP, $n \leq 10$)

- Two Components - Involving Graph

  1. *UVa 00273 - Jack Straw* (line segment intersection and Warshall's transitive closure algorithm)
  2. *UVa 00521 - Gossiping* (build a graph; the vertices are drivers; give an edge between two drivers if they can meet; this is determined with mathematical rule (gcd); if the graph is connected, then the answer is 'yes')
  3. *UVa 01039 - Simplified GSM Network* (LA 3270, World Finals Shanghai05, build the graph with simple geometry; then use Floyd Warshall's)
  4. ***UVa 01092 - Tracking Bio-bots *** (LA 4787, World Finals Harbin10, compress the graph first; do graph traversal from exit using only south and west direction; inclusion-exclusion)
  5. UVa 01243 - Polynomial-time Red... (LA 4272, Hefei08, Floyd Warshall's transitive closure, SCC, transitive reduction of a directed graph)
  6. UVa 01263 - Mines (LA 4846, Daejeon10, geometry, SCC, see two related problems: UVa 11504 & 11770)
  7. UVa 10075 - Airlines (`gcDistance`—see Section 9.11—with APSP)
  8. UVa 10307 - Killing Aliens in Borg Maze (build SSSP graph with BFS, MST)
  9. UVa 11267 - The 'Hire-a-Coder' ... (bipartite check, MST accept -ve weight)
  10. **UVa 11635 - Hotel Booking *** (Dijkstra's + BFS)
  11. UVa 11721 - Instant View ... (find nodes that can reach SCCs with neg cycle)
  12. UVa 11730 - Number Transformation (prime factoring, see Section 5.5.1)
  13. UVa 12070 - Invite Your Friends (LA 3290, Dhaka05, BFS + Dijkstra's)
  14. *UVa 12101 - Prime Path* (BFS, involving prime numbers)
  15. **UVa 12159 - Gun Fight *** (LA 4407, KualaLumpur08, geometry, MCBM)

- Two Components - Involving Mathematics

  1. *UVa 01195 - Calling Extraterrestrial ...* (LA 2565, Kanazawa02, use sieve to generate the list of primes, brute force each prime p and use binary search to find the corresponding pair q)
  2. *UVa 10325 - The Lottery* (inclusion exclusion principle, brute force subset for small $M \leq 15$, lcm-gcd)
  3. UVa 10427 - Naughty Sleepy ... (numbers in $[10^{(k-1)}..10^{k}-1]$ has k digits)

4. **UVa 10539 - Almost Prime Numbers *** (sieve; get 'almost primes': non prime numbers that are divisible by only a *single* prime number; we can get a list of 'almost primes' by listing the powers of each prime, e.g. 3 is a prime number, so $3^2 = 9$, $3^3 = 27$, $3^4 = 81$, etc are 'almost primes'; we can then sort these 'almost primes'; and then do binary search)

5. **UVa 10637 - Coprimes *** (involving prime numbers and gcd)

6. **UVa 10717 - Mint *** (complete search + GCD/LCM, see Section 5.5.2)

7. *UVa 11282 - Mixing Invitations* (derangement and binomial coefficient, use Java BigInteger)

8. *UVa 11415 - Count the Factorials* (count the number of factors for each integer, use it to find the number of factors for each factorial number and store it in an array; for each query, search in the array to find the first element with that value with binary search)

9. UVa 11428 - Cubes (complete search + binary search)

- Two Components - Complete Search and Geometry

  1. *UVa 00142 - Mouse Clicks* (brute force; point-in-rectangle; `dist`)
  2. UVa 00184 - Laser Lines (brute force; `collinear` test)
  3. UVa 00201 - Square (counting square of various sizes; try all)
  4. UVa 00270 - Lining Up (gradient sorting, complete search)
  5. UVa 00356 - Square Pegs And Round ... (Euclidean distance, brute force)
  6. *UVa 00638 - Finding Rectangles* (brute force 4 corner points)
  7. *UVa 00688 - Mobile Phone Coverage* (brute force; chop the region into small rectangles and decide if a small rectangle is covered by an antenna or not; if it is, add the area of that small rectangle to the answer)
  8. **UVa 10012 - How Big Is It? *** (try all 8! permutations, Euclidean `dist`)
  9. UVa 10167 - Birthday Cake (brute force A and B, `ccw` tests)
  10. UVa 10301 - Rings and Glue (circle-circle intersection, backtracking)
  11. UVa 10310 - Dog and Gopher (complete search, Euclidean distance `dist`)
  12. UVa 10823 - Of Circles and Squares (complete search; check if point inside circles/squares)
  13. **UVa 11227 - The silver bullet *** (brute force; `collinear` test)
  14. UVa 11515 - Cranes (circle-circle intersection, backtracking)
  15. ***UVa 11574 - Colliding Traffic **** (brute force all pairs of boats; if one pair already collide, the answer is 0.0; otherwise derive a quadratic equation to detect when these two boats will collide, if they will; pick the minimum collision time overall; if there is no collision, output 'No collision.')

- Mixed with Efficient Data Structure

  1. *UVa 00843 - Crypt Kicker* (backtracking; try mapping each letter to another letter in alphabet; use Trie data structure (see Section 6.6) to speed up if a certain (partial) word is in the dictionary)
  2. *UVa 00922 - Rectangle by the Ocean* (first, compute the area of the polygon; then for every pair of points, define a rectangle with those 2 points; use `set` to check whether a third point of the rectangle is on the polygon; check whether it is better than the current best)
  3. *UVa 10734 - Triangle Partitioning* (this is actually a geometry problem involving triangle/cosine rule, but we use a data structure that tolerates floating point imprecision due to triangle side normalization to make sure we count each triangle only once)

4. **UVa 11474 - Dying Tree *** (use union find; first, connect all branches in the tree; next, connect one tree with another tree if any of their branch has distance less than k (a bit of geometry); then, connect any tree that can reach any doctor; finally, check if the first branch of the first/sick tree is connected to any doctor; the code can be quite long; be careful)

5. **UVa 11525 - Permutation *** (can use Fenwick Tree and binary search the answer to find the lowest index i that has $RSQ(1, i) = Si$)

6. **UVa 11960 - Divisor Game *** (modified Sieve, number of divisors, static Range Maximum Query, solvable with Sparse Table data structure)

7. UVa 11966 - Galactic Bonding (use union find to keep track of the number of disjoint sets/constellations; if Euclidian dist $\leq D$, union the two stars)

8. *UVa 11967 - Hic-Hac-Hoe* (simple brute force, but we need to use C++ STL `map` as we cannot store the actual tic-tac-toe board; we only remember n coordinates and check if there are k consecutive coordinates that belong to any one player)

9. *UVa 12318 - Digital Roulette* (brute force with `set` data structure)

10. *UVa 12460 - Careful teacher* (BFS problem but needs `set` of string data structure to speed up)

- Three Components

1. **UVa 00295 - Fatman *** (binary search the answer x for this question: if the person is of diameter x, can he go from left to right? for any pair of obstacles (including the top and bottom walls), lay an edge between them if the person cannot go between them and check if the top and bottom wall are disconnected $\rightarrow$ person with diameter x can pass; Euclidian distance)

2. UVa 00811 - The Fortified Forest (LA 5211, World Finals Eindhoven99, `CH`, `perimeter` of polygon, generate all subsets iteratively with bitmask)

3. **UVa 01040 - The Traveling Judges *** (LA 3271, World Finals Shanghai05, iterative complete search, try all subsets of 2^{20} cities, form MST with those cities with help of Union-Find DS, complex output formatting)

4. UVa 01079 - A Careful Approach (LA 4445, World Finals Stockholm09, discussed in this chapter)

5. *UVa 01093 - Castles* (LA 4788, World Finals Harbin10, try all possible roots, DP on tree)

6. UVa 01250 - Robot Challenge (LA 4607, SoutheastUSA09, geometry, SSSP on DAG $\rightarrow$ DP, DP 1D range sum)

7. UVa 10856 - Recover Factorial (discussed in this section)

8. *UVa 10876 - Factory Robot* (binary search the answer + geometry, Euclidian distance + union find, similar idea with UVa 295)

9. **UVa 11610 - Reverse Prime *** (first, reverse primes less than 10^6; then, append zero(es) if necessary; use Fenwick Tree and binary search)

8.5 Solution to Non-Starred Exercises

Exercise 8.2.3.1: In C++, we can use pair<int, int> (short form: ii) to store a pair of (integer) information. For triple, we can use pair<int, ii>. For quad, we can use pair<ii, ii>. In Java, we do not have such feature and thus we have to create a class that implements comparable (so that we can use Java TreeMap to store these objects properly).

Exercise 8.2.3.2: We have no choice but to use a class in C++ and Java. For C/C++, struct is also possible. Then, we have to implement a comparison function for such class.

Exercise 8.2.3.3: State-Space Search is essentially an extension of the Single-Source *Shortest* Paths problem, which is a minimization problem. The longest path problem (maximization problem) is NP-hard and usually we do not deal with such variant as the (minimization problem of) State-Space Search is already complex enough to begin with.

Exercise 8.3.1.1: The solution is similar with UVa 10911 solution as shown in Section 1.2. But in the "Maximum Cardinality Matching" problem, there is a possibility that a vertex is *not* matched. The DP with bitmask solution for a small general graph is shown below:

```
int MCM(int bitmask) {
  if (bitmask == (1 << N) - 1)          // all vertices have been considered
    return 0;                                  // no more matching is possible
  if (memo[bitmask] != -1)
    return memo[bitmask];

  int p1, p2;
  for (p1 = 0; p1 < N; p1++)                        // find a free vertex p1
    if (!(bitmask & (1 << p1)))
      break;

  // This is the key difference:
  // We have a choice not to match free vertex p1 with anything
  int ans = MCM(bitmask | (1 << p1));

  // Assume that the small graph is stored in an Adjacency Matrix AdjMat
  for (p2 = 0; p2 < N; p2++)    // find neighbors of vertex p1 that are free
    if (AdjMat[p1][p2] && p2 != p1 && !(bitmask & (1 << p2)))
      ans = max(ans, 1 + MCM(bitmask | (1 << p1) | (1 << p2)));

  return memo[bitmask] = ans;
}
```

Exercise 8.4.8.1: Please refer to Section 3.3.1 for the solution.

8.6 Chapter Notes

We have significantly improve this Chapter 8 as promised in the chapter notes of the previous (second) edition. In the third edition, this Chapter 8 roughly has twice the number of pages and exercises because of two reasons. First: We have solved quite a number of harder problems in between the second and the third edition. Second: We have moved some of the harder problems that were previously listed in the earlier chapters (in the second edition) into this chapter—most notably from Chapter 7 (into Section 8.4.6).

In the third edition, this Chapter 8 is no longer the last chapter. We still have one more Chapter 9 where we list down rare topics that rarely appears, but may be of interest for enthusiastic problem solvers.

Statistics	First Edition	Second Edition	Third Edition
Number of Pages	-	15	33 (+120%)
Written Exercises	-	3	5+8*=13 (+333%)
Programming Exercises	-	83	177 (+113%)

The breakdown of the number of programming exercises from each section is shown below:

Section	Title	Appearance	% in Chapter	% in Book
8.2	More Advanced Search	36	20%	2%
8.3	**More Advanced DP**	51	29%	3%
8.4	**Problem Decomposition**	90	51%	5%

—The first time both of us attended ACM ICPC World Finals together.—

Chapter 9

Rare Topics

> *Learning is a treasure that will follow its owner everywhere.*
> — **Chinese Proverb**

Overview and Motivation

In this chapter, we list down rare, 'exotic' topics in Computer Science that may (but usually will not) appear in a typical programming contest. These problems, data structures, and algorithms are mostly one-off unlike the more general topics that have been discussed in Chapters 1-8. Learning the topics in this chapter can be considered as being not 'cost-efficient' because after so much efforts on learning a certain topic, it likely *not* appear in the programming contest. However, we believe that these rare topics will appeal those who really love to expand their knowledge in Computer Science.

Skipping this chapter will not cause a major damage towards the preparation for an ICPC-style programming contest as the probability of appearance of any of these topics is low anyway[1]. However, when these rare topics do appear, contestants with a priori knowledge of those rare topics will have an advantage over others who do not have such knowledge. Some good contestants can probably derive the solution from basic concepts during contest time even if they have only seen the problem for the first time, but usually in a slower pace than those who already know the problem and especially its solution before.

For IOI, many of these rare topics are outside the IOI syllabus [20]. Thus, IOI contestants can choose to defer learning the material in this chapter until they enroll in University.

In this chapter, we keep the discussion for each topic as concise as possible, i.e. most discussions will be just around one or two page(s). Most discussions do not contain sample code as readers who have mastered the content of Chapter 1-8 should not have too much difficulty in translating the algorithms given in this chapter into a working code. We only have a few starred written exercises (without hints/solutions) in this chapter.

These rare topics are listed in alphabetical order in the table of contents at the front of this book. However, if you cannot find the name that we use, please use the indexing feature at the back of this book to check if the alternative names of these topics are listed.

---

[1]Some of these topics—with low probability—are used as interview questions for IT companies.

9.1   2-SAT Problem

Problem Description

You are given a conjunction of disjunctions ("and of ors") where each disjunction ("the or operation") has two arguments that may be variables or the negation of variables. The disjunctions of pairs are called as 'clauses' and the formula is known as the 2-CNF (Conjunctive Normal Form) formula. The 2-SAT problem is to find a truth (that is, true or false) assignment to these variables that makes the 2-CNF formula true, i.e. every clause has at least one term that is evaluated to true.

Example 1: $(x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2)$ is satisfiable because we can assign $x_1 = true$ and $x_2 = false$ (alternative assignment is $x_1 = false$ and $x_2 = true$).

Example 2: $(x_1 \vee x_2) \wedge (\neg x_1 \vee x_2) \wedge (\neg x_2 \vee x_3) \wedge (\neg x_2 \vee \neg x_3)$ is not satisfiable. You can try all 8 possible combinations of boolean values of x_1, x_2, and x_3 to realize that none of them can make the 2-CNF formula satisfiable.

Solution(s)

Complete Search

Contestants who only have a vague knowledge of the Satisfiability problem may thought that this problem is an NP-Complete problem and therefore attempt a complete search solution. If the 2-CNF formula has n variables and m clauses, trying all 2^n possible assignments and checking each assignment in $O(m)$ has an overall time complexity of $O(2^n \times m)$. This is likely TLE.

The 2-SAT is a *special case* of Satisfiability problem and it admits a polynomial solution like the one shown below.

Reduction to Implication Graph and Finding SCC

First, we have to realize that a clause in a 2-CNF formula $(a \vee b) \equiv (\neg a \Rightarrow b) \equiv (\neg b \Rightarrow a)$. Thus, given a 2-CNF formula, we can build the corresponding 'implication graph'. Each variable has two vertices in the implication graph, the variable itself and the negation/inverse of that variable[2]. An edge connects one vertex to another if the corresponding variables are related by an implication in the corresponding 2-CNF formula. For the two 2-CNF example formulas above, we have the following implication graphs shown in Figure 9.1.
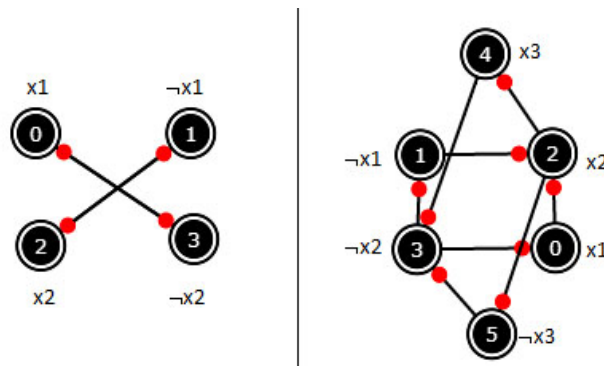
Figure 9.1: The Implication Graph of Example 1 (Left) and Example 2 (Right)

---

[2]Programming trick: We give a variable an index i and its negation with another index $i + 1$. This way, we can find one from the other by using bit manipulation $i \oplus 1$ where $\oplus$ is the 'exclusive or' operator.

As you can see in Figure 9.1, a 2-CNF formula with n variables (excluding the negation) and m clauses will have $V = \theta(2n) = \theta(n)$ vertices and $E = O(2m) = O(m)$ edges in the implication graph.

Now, a 2-CNF formula is satisfiable if and only if "there is no variable that belongs to the same Strongly Connected Component (SCC) as its negation".

In Figure 9.1—left, we see that there are two SCCs: {0,3} and {1,2}. As there is no variable that belongs to the same SCC as its negation, we conclude that the 2-CNF formula shown in Example 1 is satisfiable.

In Figure 9.1—right, we observe that all six vertices belong to a single SCC. Therefore, we have vertex 0 (that represents x_1) and vertex 1 (that represents[3] $\neg x_1$), vertex 2 (x_2) and vertex 3 ($\neg x_3$), vertex 4 (x_3) and vertex 5 ($\neg x_3$) in the same SCC. Therefore, we conclude that the 2-CNF formula shown in Example 2 is not satisfiable.

To find the SCCs of a directed graph, we can use either Tarjan's SCC algorithm as shown in Section 4.2.9 or Kosaraju's SCC algorithm as shown in Section 9.17.

Exercise 9.1.1*: To find the actual truth assignment, we need to do a bit more work than just checking if there is no variable that belongs to the same SCC as its negation. What are the extra steps required to actually find the truth assignment of a satisfiable 2-CNF formula?

Programming exercises related to 2-SAT problem:

1. **_UVa 10319 - Manhattan *_** (the hard part in solving problems involving 2-SAT is in identifying that it is indeed a 2-SAT problem and then building the implication graph; for this problem, we set each street and each avenue as a variable where true means that it can only be used in a certain direction and false means that it can only be used in the other direction; a simple path will be in one of this form: (street a ∧ avenue b) ∨ (avenue c ∧ street d); this can be transformed into 2-CNF formula of $(a \lor c) \land (a \lor d) \land (b \lor c) \land (b \lor d)$; build the implication graph and check if it is satisfiable using the SCC check as shown above; note that there exists a special case where the clause only has one literal, i.e. the simple path uses one street only or one avenue only.)

---

[3]Notice that using the programming trick shown above, we can easily test if vertex 1 and vertex 0 are a variable and its negation by testing if $1 = 0 \oplus 1$.

9.2 Art Gallery Problem

Problem Description

The 'Art Gallery' Problem is a family of related *visibility* problems in computational geometry. In this section, we discuss several variants. The common terms used in the variants discussed below are the simple (not necessarily convex) polygon P to describe the art gallery; a set of points S to describe the guards where each guard is represented by a point in P; a rule that a point $A \in S$ can guard another point $B \in P$ if and only if $A \in S, B \in P$, and line segment AB is contained in P; and a question on whether all points in polygon P are guarded by S. Many variants of this Art Gallery Problem are classified as NP-hard problems. In this book, we focus on the ones that admit polynomial solutions.

1. Variant 1: Determine the upper bound of the smallest size of set S.

2. Variant 2: Determine if $\exists$ a critical point C in polygon P and $\exists$ another point $D \in P$ such that if the guard is at position C, the guard cannot protect point D.

3. Variant 3: Determine if polygon P can be guarded with just one guard.

4. Variant 4: Determine the smallest size of set S if the guards can only be placed at the vertices of polygon P and only the vertices need to be guarded.

Note that there are many more variants and at least one book[4] has been written for it [49].

Solution(s)

1. The solution for variant 1 is a theoretical work of the Art Gallery theorem by Václav Chvátal. He states that $\lfloor n/3 \rfloor$ guards are always sufficient and sometimes necessary to guard a simple polygon with n vertices (proof omitted).

2. The solution for variant 2 involves testing if polygon P is concave (and thus has a critical point). We can use the negation of `isConvex` function shown in Section 7.3.4.

3. The solution for variant 3 can be hard if one has not seen the solution before. We can use the `cutPolygon` function discussed in Section 7.3.6. We cut polygon P with all lines formed by the edges in P in counter clockwise fashion and retain the left side at all times. If we still have a non empty polygon at the end, one guard can be placed in that non empty polygon which can protect the entire polygon P.

4. The solution for variant 4 involves the computation of Minimum Vertex Cover of the 'visibility graph' of polygon P. In general this is another NP-hard problem.

Programming exercises related to Art Gallery problem:

1. **_UVa 00588 - Video Surveillance_ *** (see variant 3 solution above)
2. **UVa 10078 - Art Gallery *** (see variant 2 solution above)
3. **UVa 10243 - Fire; Fire; Fire *** (variant 4: this problem can be reduced to the Minimum Vertex Cover problem *on Tree*; there is a polynomial DP solution for this variant; the solution has actually been discussed Section 4.7.1)
4. LA 2512 - Art Gallery (see variant 3 solution above plus area of polygon)
5. LA 3617 - How I Mathematician ... (variant 3)

---

[4]PDF version at `http://cs.smith.edu/~orourke/books/ArtGalleryTheorems/art.html`.

9.3 Bitonic Traveling Salesman Problem

Problem Description

The Bitonic Traveling Salesman Problem (TSP) can be described as follows: Given a list of coordinates of n vertices on 2D Euclidean space that are already sorted by x-coordinates (and if tie, by y-coordinates), find a tour that starts from the leftmost vertex, then goes strictly from left to right, and then upon reaching the rightmost vertex, the tour goes strictly from right to left back to the starting vertex. This tour behavior is called 'bitonic'.

The resulting tour may not be the shortest possible tour under the standard definition of TSP (see Section 3.5.2). Figure 9.2 shows a comparison of these two TSP variants. The TSP tour: 0-3-5-6-4-1-2-0 is not a Bitonic TSP tour because although the tour initially goes from left to right (0-3-5-6) and then goes back from right to left (6-4-1), it then makes another left to right (1-2) and then right to left (2-0) steps. The tour: 0-2-3-5-6-4-1-0 is a valid Bitonic TSP tour because we can decompose it into two paths: 0-2-3-5-6 that goes from left to right and 6-4-1-0 that goes back from right to left.
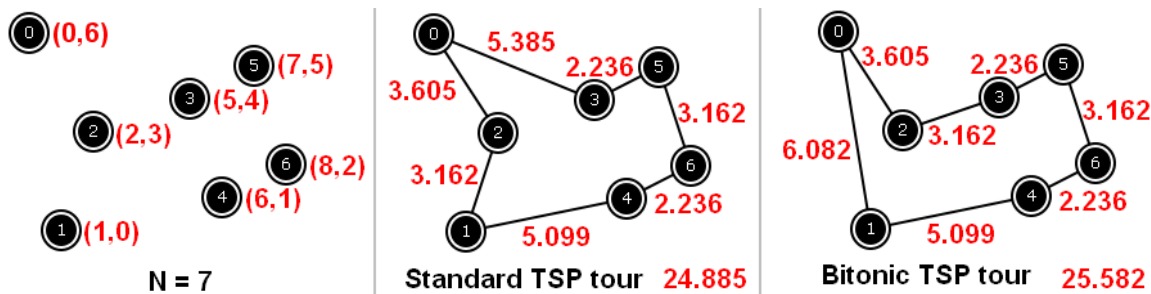
Figure 9.2: The Standard TSP versus Bitonic TSP

Solution(s)

Although a Bitonic TSP tour of a set of n vertices is usually longer than the standard TSP tour, this bitonic constraint allows us to compute a 'good enough tour' in $O(n^2)$ time using Dynamic Programming—as shown below—compared with the $O(2^n \times n^2)$ time for the standard TSP tour (see Section 3.5.2).

The main observation needed to derive the DP solution is the fact that we can (and have to) split the tour into two paths: Left-to-Right (LR) and Right-to-Left (RL) paths. Both paths include vertex 0 (the leftmost vertex) and vertex n-1 (the rightmost vertex). The LR path starts from vertex 0 and ends at vertex n-1. The RL path starts from vertex n-1 and ends at vertex 0.

Remember that all vertices have been sorted by x-coordinates (and if tie, by y-coordinates). We can then consider the vertices one by one. Both LR and RL paths start from vertex 0. Let v be the next vertex to be considered. For each vertex $v \in [1 \ldots n-2]$, we decide whether to add vertex v as the next point of the LR path (to extend the LR path further to the right) or as the previous point the returning RL path (the RL path now starts at v and goes back to vertex 0). For this, we need to keep track of two more parameters: $p1$ and $p2$. Let $p1/p2$ be the current *ending/starting* vertex of the LR/RL path, respectively.

The base case is when vertex $v = n-1$ where we just need to connect the two LR and RL paths with vertex $n-1$.

With these observations in mind, we can write a simple DP solution is like this:

```
double dp1(int v, int p1, int p2) {                  // called with dp1(1, 0, 0)
  if (v == n-1)
    return d[p1][v] + d[v][p2];
  if (memo3d[v][p1][p2] > -0.5)
    return memo3d[v][p1][p2];
  return memo3d[v][p1][p2] = min(
    d[p1][v] + dp1(v+1, v, p2),        // extend LR path: p1->v, RL stays: p2
    d[v][p2] + dp1(v+1, p1, v));       // LR stays: p1, extend RL path: p2<-v
}
```

However, the time complexity of `dp1` with three parameters: (`v`, `p1`, `p2`) is $O(n^3)$. This is not efficient, as parameter v can be dropped and recovered from $1 + max(p1, p2)$ (see this DP optimization technique of dropping one parameter and recovering it from other parameters as shown in Section 8.3.6). The improved DP solution is shown below and runs in $O(n^2)$.

```
double dp2(int p1, int p2) {                            // called with dp2(0, 0)
  int v = 1 + max(p1, p2);    // this single line speeds up Bitonic TSP tour
  if (v == n-1)
    return d[p1][v] + d[v][p2];
  if (memo2d[p1][p2] > -0.5)
    return memo2d[p1][p2];
  return memo2d[p1][p2] = min(
    d[p1][v] + dp2(v, p2),            // extend LR path: p1->v, RL stays: p2
    d[v][p2] + dp2(p1, v));           // LR stays: p1, extend RL path: p2<-v
}
```

Programming exercises related to Bitonic TSP:

1. **UVa 01096 - The Islands *** (LA 4791, World Finals Harbin10, Bitonic TSP variant; print the actual path)

2. ***UVa 01347 - Tour **** (LA 3305, Southeastern Europe 2005; this is the pure version of Bitonic TSP problem, you may want to start from here)

9.4 Bracket Matching

Problem Description

Programmers are very familiar with various form of braces: '()', '{}', '[]', etc as they use braces quite often in their code especially when dealing with if statements and loops. Braces can be nested, e.g. '(())', '{{}}', '[[]]', etc. A well-formed code must have a matched set of braces. The Bracket Matching problem usually involves a question on whether a given set of braces is properly nested. For example, '(())', '({})', '(){}[]' are correctly matched braces whereas '(()', '(}', ')(' are *not* correct.

Solution(s)

We read the brackets one by one from left to right. Every time we encounter a close bracket, we need to match it with the latest open bracket (of the same type). This matched pair is then removed from consideration and the process is continued. This requires a 'Last In First Out' data structure: Stack (see Section 2.2).

We start from an empty stack. Whenever we encounter an open bracket, we push it into the stack. Whenever we encounter a close bracket, we check if it is of the same type with the top of the stack. This is because the top of the stack is the one that has to be matched with the current close bracket. Once we have a match, we pop the topmost bracket from the stack to remove it from future consideration. Only if we manage to reach the last bracket and find that the stack is back to empty, then we know that the brackets are properly nested.

As we examine each of the n braces only once and all stack operations are $O(1)$, this algorithm clearly runs in $O(n)$.

Variant(s)

The number of ways n pairs of parentheses can be correctly matched can be found with Catalan formula (see Section 5.4.3). The optimal way to multiply matrices (i.e. the Matrix Chain Multiplication problem) also involves bracketing. This variant can be solved with Dynamic Programming (see Section 9.20).

Programming exercises related to Bracket Matching:

1. ***UVa 00551 - Nesting a Bunch of ... **** (bracket matching, `stack`, classic)
2. **UVa 00673 - Parentheses Balance *** (similar to UVa 551, classic)
3. ***UVa 11111 - Generalized Matrioshkas **** (bracket matching with some twists)

9.5 Chinese Postman Problem

Problem Description

The Chinese Postman[5]/Route Inspection Problem is the problem of finding the (length of the) shortest tour/circuit that visits every edge of a (connected) undirected weighted graph. If the graph is Eulerian (see Section 4.7.3), then the sum of edge weights along the Euler tour that covers all the edges in the Eulerian graph is the optimal solution for this problem. This is the easy case. But when the graph is non Eulerian, e.g. see the graph in Figure 9.3—left, then this Chinese Postman Problem is harder.

Solution(s)

The important insight to solve this problem is to realize that a non Eulerian graph G must have an *even number* of vertices of odd degree (the Handshaking lemma found by Euler himself). Let's name the subset of vertices of G that have odd degree as T. Now, create a complete graph K_n where n is the size of T. T form the vertices of K_n. An edge (i, j) in K_n has weight which is the *shortest path weight* of a path from i to j, e.g. in Figure 9.3 (middle), edge 2-5 in K_4 has weight $2 + 1 = 3$ from path 2-4-5 and edge 3-4 in K_4 has weight $3 + 1 = 4$ from path 3-5-4.
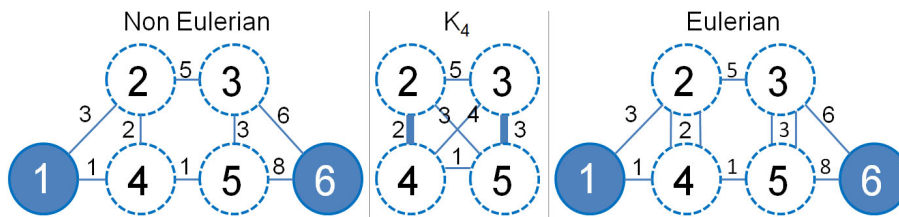
Figure 9.3: An Example of Chinese Postman Problem

Now, if we *double* the edges selected by the *minimum weight perfect matching* on this complete graph K_n, we will convert the non Eulerian graph G to another graph G' which is Eulerian. This is because by doubling those edges, we actually add an edge between a pair of vertices with odd degree (thus making them have even degree afterwards). The *minimum weight* perfect matching ensures that this transformation is done in the *least cost way*. The solution for the minimum weight perfect matching on the K_4 shown in Figure 9.3 (middle) is to take edge 2-4 (with weight 2) and edge 3-5 (with weight 3).

After doubling edge 2-4 and edge 3-5, we are now back to the easy case of the Chinese Postman Problem. In Figure 9.3 (right), we have an Eulerian graph. The tour is simple in this Eulerian graph. One such tour is: 1->2->4->5->3->6->5->3->2->4->1 with a total weight of 34 (the sum of all edge weight in the modified Eulerian graph G', which is the sum of all edge weight in G plus the cost of the minimum weight perfect matching in K_n).

The hardest part of solving the Chinese Postman Problem is therefore in finding the minimum weight perfect matching on K_n, which is *not* a bipartite graph (a complete graph). If n is small, this can be solved with DP with bitmask technique shown in Section 8.3.1.

Programming exercises related to Chinese Postman Problem:

1. **UVa 10296 - Jogging Trails** * (see the discussion above)

---

[5]The name is because it is first studied by the Chinese mathematician Mei-Ku Kuan in 1962.

9.6   Closest Pair Problem

Problem Description

Given a set S of n points on a 2D plane, find two points with the closest Euclidean distance.

Solution(s)

Complete Search

A naïve solution computes the distances between all pairs of points and reports the minimum one. However, this requires $O(n^2)$ time.

Divide and Conquer

We can use the following Divide and Conquer strategy to achieve $O(n \log n)$ time.
We perform the following three steps:

1. Divide: We sort the points in set S by their x-coordinates (if tie, by their y-coordinates). Then, we divide set S into two sets of points S_1 and S_2 with a vertical line $x = d$ such that $|S_1| = |S_2|$ or $|S_1| = |S_2| + 1$, i.e. the number of points in each set is balanced.

2. Conquer: If we only have one point in S, we return ∞. If we only have two points in S, we return their Euclidean distance.

3. Combine: Let d_1 and d_2 be the smallest distance in S_1 and S_2, respectively. Let d_3 be the smallest distance between all pairs of points (p_1, p_2) where p_1 is a point in S_1 and p_2 is a point in S_2. Then, the smallest distance is $min(d_1, d_2, d_3)$, i.e. the answer may be in the smaller set of points S_1 or in S_2 or one point in S_1 and the other point in S_2, crossing through line $x = d$.

The combine step, if done naïvely, will still run in $O(n^2)$. But this can be optimized. Let $d' = min(d_1, d_2)$. For each point in the left of the dividing line $x = d$, a closer point in the right of the dividing line can only lie within a rectangle with width d' and height $2 \times d'$. It can be proven (proof omitted) that there can be only at most 6 such points in this rectangle. This means that the combine step only require $O(6n)$ operation and the overall time complexity of this divide and conquer solution is $T(n) = 2 \times T(n/2) + O(n)$ which is $O(n \log n)$.

Exercise 9.6.1*: There is a simpler solution other than the classic Divide & Conquer solution shown above. It uses sweep line algorithm. We 'sweep' the points in S from left to right. Suppose the current best answer is d and we are now examining point i. The potential new closest point from i, if any, must have y-coordinate to be within d units of point i. We check all these candidates and update d accordingly (which will be progressively smaller). Implement this solution and analyze its time complexity!

Programming exercises related to Closest Pair problem:

1. **UVa 10245 - The Closest Pair Problem *** (classic, as discussed above)
2. **UVa 11378 - Bey Battle *** (also a closest pair problem)

9.7 Dinic's Algorithm

In Section 4.6, we have seen the potentially unpredictable $O(|f^*|E)$ Ford Fulkerson's method and the preferred $O(VE^2)$ Edmonds Karp's algorithm (finding augmenting paths with BFS) for solving the Max Flow problem. As of year 2013, most (if not all) Max Flow problems in this book are solvable using Edmonds Karp's.

There are several other Max Flow algorithms that have theoretically better performance than Edmonds Karp's. One of them is Dinic's algorithm which runs in $O(V^2E)$. Since a typical flow graph usually has $V < E$ and $E << V^2$, Dinic's worst case time complexity is theoretically better than Edmonds Karp's. Although the authors of this book have not encountered a case where Edmonds Karp's received TLE verdict and Dinic's received AC verdict on the *same* flow graph, it *may be* beneficial to use Dinic's algorithm in programming contests just to be on the safer side.

Dinic's algorithm uses a similar idea as Edmonds Karp's as it also finds augmenting paths iteratively. However, Dinic's algorithm uses the concept of 'blocking flows' to find the augmenting paths. Understanding this concept is the key to extend the easier-to-understand Edmonds Karp's algorithm into Dinic's algorithm.

Let's define `dist[v]` to be the length of the shortest path from the source vertex s to v in the residual graph. Then the level graph of the residual graph is L where edge (u, v) in the residual graph is included in the level graph L iff `dist[v] = dist[u] + 1`. Then, a 'blocking flow' is an $s-t$ flow f such that after sending through flow f from s to t, the level graph L contains no $s-t$ augmenting path anymore.

It has been proven (see [11]) that the number of edges in each blocking flow increases by at least one per iteration. There are at most $V-1$ blocking flows in the algorithm because there can only be at most $V-1$ edges along the 'longest' simple path from s to t. The level graph can be constructed by a BFS in $O(E)$ time and a blocking flow in each level graph can be found in $O(VE)$ time. Hence, the worst case time complexity of Dinic's algorithm is $O(V \times (E + VE)) = O(V^2E)$.

Dinic's implementation can be made similar with Edmonds Karp's implementation shown in Section 4.6. In Edmonds Karp's, we run a BFS—which already generates for us the level graph L—but we just use it to find *one* augmenting path by calling `augment(t, INF)` function. In Dinic's algorithm, we need to use the information produced by BFS in a slightly different manner. The key change is this: Instead of finding a blocking flow by running DFS on the level graph L, we can simulate the process by running the `augment` procedure from *each vertex v that is directly connected to the sink vertex t* in the level graph, i.e. edge (v, t) exists in level graph L and we call `augment(v, INF)`. This will find many (but not always all) the required augmenting *paths* that make up the blocking flow of level graph L for us.

Exercise 9.7.1*: Implement the *variant* of Dinic's algorithm *s*tarting from the Edmonds Karp's code given in Section 4.6 using the suggested key change above! Also use the modified Adjacency List as asked in **Exercise 4.6.3.3***. Now, (re)solve various programming exercises listed in Section 4.6! Do you notice any runtime improvements?

Exercise 9.7.2*: Using a data structure called dynamic trees, the running time of finding a blocking flow can be reduced from $O(VE)$ down to $O(E \log V)$ and therefore the overall worst-case time complexity of Dinic's algorithm becomes $O(VE \log V)$. Study and implement this Dinic's implementation variant!

Exercise 9.7.3*: What happens if we use Dinic's algorithm on flow graph that models MCBM problem as discussed in Section 4.7.4? (hint: See Section 9.12).

9.8   Formulas or Theorems

We have encountered some rarely used formulas or theorems in some programming contest problems before. Knowing them will give you an *unfair advantage* over other contestants if one of these rare formulas or theorems is used in the programming contest that you join.

1. Cayley's Formula: There are n^{n-2} spanning trees of a complete graph with n labeled vertices. Example: UVa 10843 - Anne's game.

2. Derangement: A permutation of the elements of a set such that none of the elements appear in their original position. The number of derangements $der(n)$ can be computed as follow: $der(n) = (n-1) \times (der(n-1) + der(n-2))$ where $der(0) = 1$ and $der(1) = 0$. A basic problem involving derangement is UVa 12024 - Hats (see Section 5.6).

3. Erdős Gallai's Theorem gives a necessary and sufficient condition for a finite sequence of natural numbers to be the *degree sequence* of a simple graph. A sequence of non-negative integers $d_1 \geq d_2 \geq \ldots \geq d_n$ can be the degree sequence of a simple graph on n vertices iff $\sum_{i=1}^{n} d_i$ is even and $\sum_{i=1}^{k} d_i \leq k \times (k-1) + \sum_{i=k+1}^{n} min(d_i, k)$ holds for $1 \leq k \leq n$. Example: UVa 10720 - Graph Construction.

4. Euler's Formula for Planar Graph[6]: $V - E + F = 2$, where F is the number of faces[7] of the Planar Graph. Example: UVa 10178 - Count the Faces.

5. Moser's Circle: Determine the number of pieces into which a circle is divided if n points on its circumference are joined by chords with no three internally concurrent. Solution: $g(n) = {}^{n}C_4 + {}^{n}C_2 + 1$. Example: UVa 10213 - How Many Pieces of Land?

6. Pick's Theorem[8]: Let I be the number of integer points in the polygon, A be the area of the polygon, and b be the number of integer points on the boundary, then $A = i + \frac{b}{2} - 1$. Example: UVa 10088 - Trees on My Island.

7. The number of spanning tree of a complete bipartite graph $K_{n,m}$ is $m^{n-1} \times n^{m-1}$. Example: UVa 11719 - Gridlands Airport.

Programming exercises related to *rarely used* Formulas or Theorems:

1. UVa 10088 - Trees on My Island (Pick's Theorem)

2. UVa 10178 - Count the Faces (Euler's Formula, a bit of union find)

3. **UVa 10213 - How Many Pieces ... *** (Moser's circle; the formula is hard to derive; $g(n) = {}_{n}C_4 + {}_{n}C_2 + 1$)

4. **UVa 10720 - Graph Construction *** (Erdős-Gallai's Theorem)

5. UVa 10843 - Anne's game (Cayley's Formula to count the number of spanning trees of a graph with n vertices is n^{n-2}; use Java BigInteger)

6. UVa 11414 - Dreams (similar to UVa 10720; Erdős-Gallai's Theorem)

7. ***UVa 11719 - Gridlands Airports **** (count the number of spanning tree in a complete bipartite graph; use Java BigInteger)

---

[6]Graph that can be drawn on 2D Euclidean space so that no two edges in the graph cross each other.
[7]When a Planar Graph is drawn without any crossing, any cycle that surrounds a region without any edges reaching from the cycle into the region forms a face.
[8]Found by Georg Alexander Pick.

9.9 Gaussian Elimination Algorithm

Problem Description

A **linear equation** is defined as an equation where the order of the unknowns (variables) is **linear** (a constant or a product of a constant plus the first power of an unknown). For example, equation X + Y = 2 is linear but equation $X^2 = 4$ is not linear.

A **system of linear equations** is defined as a collection of n unknowns (variables) in (usually) n linear equations, e.g. X + Y = 2 and 2X + 5Y = 6, where the solution is X = $1\frac{1}{3}$, Y = $\frac{2}{3}$. Notice the difference to the **linear diophantine equation** (see Section 5.5.9) as the solution for a **system of linear equations** can be non-integers!

In rare occasions, we may find such system of linear equations in a programming contest problem. Knowing the solution, especially its implementation, may come handy.

Solution(s)

To compute the solution of a **system of linear equations**, one can use techniques like the **Gaussian Elimination** algorithm. This algorithm is more commonly found in Engineering textbooks under the topic of 'Numerical Methods'. Some Computer Science textbooks do have some discussions about this algorithm, e.g. [8]. Here, we show this relatively simple $O(n^3)$ algorithm using a C++ function below.

```cpp
#define MAX_N 100                              // adjust this value as needed
struct AugmentedMatrix { double mat[MAX_N][MAX_N + 1]; };
struct ColumnVector { double vec[MAX_N]; };

ColumnVector GaussianElimination(int N, AugmentedMatrix Aug) {    // O(N^3)
  // input: N, Augmented Matrix Aug, output: Column vector X, the answer
  int i, j, k, l; double t; ColumnVector X;

  for (j = 0; j < N - 1; j++) {              // the forward elimination phase
    l = j;
    for (i = j + 1; i < N; i++)        // which row has largest column value
      if (fabs(Aug.mat[i][j]) > fabs(Aug.mat[l][j]))
        l = i;                                       // remember this row l
    // swap this pivot row, reason: to minimize floating point error
    for (k = j; k <= N; k++)              // t is a temporary double variable
      t = Aug.mat[j][k], Aug.mat[j][k] = Aug.mat[l][k], Aug.mat[l][k] = t;
    for (i = j + 1; i < N; i++)      // the actual forward elimination phase
      for (k = N; k >= j; k--)
        Aug.mat[i][k] -= Aug.mat[j][k] * Aug.mat[i][j] / Aug.mat[j][j];
  }

  for (j = N - 1; j >= 0; j--) {              // the back substitution phase
    for (t = 0.0, k = j + 1; k < N; k++) t += Aug.mat[j][k] * X.vec[k];
    X.vec[j] = (Aug.mat[j][N] - t) / Aug.mat[j][j];   // the answer is here
  }
  return X;
}
```

Source code: `GaussianElimination.cpp/java`

Sample Execution

In this subsection, we show the step-by-step working of 'Gaussian Elimination' algorithm using the following example. Suppose we are given this system of linear equations:

```
X = 9 - Y - 2Z
2X + 4Y = 1 + 3Z
3X - 5Z = -6Y
```

First, we need to transform the system of linear equations into the *basic form*, i.e. we reorder the unknowns (variables) in sorted order on the Left Hand Side. We now have:

```
1X + 1Y + 2Z = 9
2X + 4Y - 3Z = 1
3X + 6Y - 5Z = 0
```

Then, we re-write these linear equations as matrix multiplication: $A \times x = b$. This trick is also used in Section 9.21. We now have:

$$\begin{bmatrix} 1 & 1 & 2 \\ 2 & 4 & -3 \\ 3 & 6 & -5 \end{bmatrix} \times \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} 9 \\ 1 \\ 0 \end{bmatrix}$$

Later, we will work with both matrix A (of size $N \times N$) and column vector b (of size $N \times 1$). So, we combine them into an $N \times (N + 1)$ 'augmented matrix' (the last column that has three arrows is a comment to aid the explanation):

$$\begin{bmatrix} 1 & 1 & 2 & 9 \\ 2 & 4 & -3 & 1 \\ 3 & 6 & -5 & 0 \end{bmatrix} \begin{array}{l} \rightarrow 1X + 1Y + 2Z = 9 \\ \rightarrow 2X + 4Y - 3Z = 1 \\ \rightarrow 3X + 6Y - 5Z = 0 \end{array}$$

Then, we pass this augmented matrix into Gaussian Elimination function above. The first phase is the forward elimination phase. We pick the largest absolute value in column $j = 0$ from row $i = 0$ onwards, then swap that row with row $i = 0$. This (extra) step is just to minimize floating point error. For this example, after swapping row 0 with row 2, we have:

$$\begin{bmatrix} \underline{3} & 6 & -5 & 0 \\ 2 & 4 & -3 & 1 \\ \underline{1} & 1 & 2 & 9 \end{bmatrix} \begin{array}{l} \rightarrow \underline{3X + 6Y - 5Z = 0} \\ \rightarrow 2X + 4Y - 3Z = 1 \\ \rightarrow \underline{1X + 1Y + 2Z = 9} \end{array}$$

The main action done by Gaussian Elimination algorithm in this forward elimination phase is to eliminate variable X (the first variable) from row $i + 1$ onwards. In this example, we eliminate X from row 1 and row 2. Concentrate on the comment "the actual forward elimination phase" inside the Gaussian Elimination code above. We now have:

$$\begin{bmatrix} 3 & 6 & -5 & 0 \\ 0 & 0 & 0.33 & 1 \\ 0 & -1 & 3.67 & 9 \end{bmatrix} \begin{array}{l} \rightarrow \underline{3X + 6Y - 5Z = 0} \\ \rightarrow \underline{0X + 0Y + 0.33Z = 1} \\ \rightarrow \underline{0X - 1Y + 3.67Z = 9} \end{array}$$

Then, we continue eliminating the next variable (now variable Y). We pick the largest absolute value in column $j = 1$ from $row = 1$ onwards, then swap that row with row $i = 1$. For this example, after swapping row 1 with row 2, we have the following augmented matrix and it happens that variable Y is already eliminated from row 2:

$$\begin{bmatrix} \text{row 0} & 3 & 6 & -5 & | & 0 \\ \text{row 1} & 0 & -1 & 3.67 & | & 9 \\ \text{row 2} & 0 & 0 & 0.33 & | & 1 \end{bmatrix} \begin{matrix} \rightarrow 3X + 6Y - 5Z = 0 \\ \rightarrow \underline{0X - 1Y + 3.67Z = 9} \\ \rightarrow \underline{0X + 0Y + 0.33Z = 1} \end{matrix}$$

Once we have the lower triangular matrix of the augmented matrix all zeroes, we can start the second phase: The back substitution phase. Concentrate on the last few lines in the Gaussian Elimination code above. Notice that after eliminating variable X and Y, there is only variable Z in row 2. We are now sure that $Z = 1/0.33 = 3$.

$$\begin{bmatrix} \text{row 2} & 0 & 0 & 0.33 & | & 1 \end{bmatrix} \rightarrow 0X + 0Y + 0.33Z = 1 \rightarrow Z = 1/0.33 = 3$$

Once we have $Z = 3$, we can process row 1.
We get $Y = (9 - 3.67 * 3)/ - 1 = 2$.

$$\begin{bmatrix} \text{row 1} & 0 & -1 & 3.67 & | & 9 \end{bmatrix} \rightarrow 0X - 1Y + 3.67Z = 9 \rightarrow Y = (9 - 3.67 * 3) / -1 = 2$$

Finally, once we have $Z = 3$ and $Y = 2$, we can process row 0.
We get $X = (0 - 6 * 2 + 5 * 3)/3 = 1$, done!

$$\begin{bmatrix} \text{row 0} & 3 & 6 & -5 & | & 0 \end{bmatrix} \rightarrow 3X + 6Y - 5Z = 0 \rightarrow X = (0 - 6 * 2 + 5 * 3) / 3 = 1$$

Therefore, the solution for the given system of linear equations is $X = 1$, $Y = 2$, and $Z = 3$.

Programming Exercises related to Gaussian Elimination:

1. **UVa 11319 - Stupid Sequence?** * (solve the system of the first 7 linear equations; then use all 1500 equations for 'smart sequence' checks)

9.10   Graph Matching

Problem Description

Graph matching: Select a subset of edges M of a graph $G(V, E)$ so that no two edges share the same vertex. Most of the time, we are interested in the *Maximum Cardinality* matching, i.e. we want to know the *maximum number of edges* that we can match in graph G. Another common request is the *Perfect* matching where we have both Maximum Cardinality matching and no vertex is left unmatched. Note that if V is odd, it is impossible to have a Perfect matching. Perfect matching can be solved by simply looking for Maximum Cardinality and then checking if all vertices are matched.

There are two important attributes of graph matching problems in programming contests that can (significantly) alter the level of difficulty: Whether the input graph is bipartite (harder otherwise) and whether the input graph is unweighted (harder otherwise). This two characteristics create four variants[9] as outlined below (also see Figure 9.4).

1. Unweighted Maximum Cardinality Bipartite Matching (Unweighted MCBM)
   This is the easiest and the most common variant.

2. Weighted Maximum Cardinality Bipartite Matching (Weighted MCBM)
   This is a similar problem as above, but now the edges in G have weights.
   We usually want the MCBM with the minimum total weight.

3. Unweighted Maximum Cardinality Matching (Unweighted MCM)
   The graph is not guaranteed to be bipartite.

4. Weighted Maximum Cardinality Matching (Weighted MCM)
   This is the hardest variant.
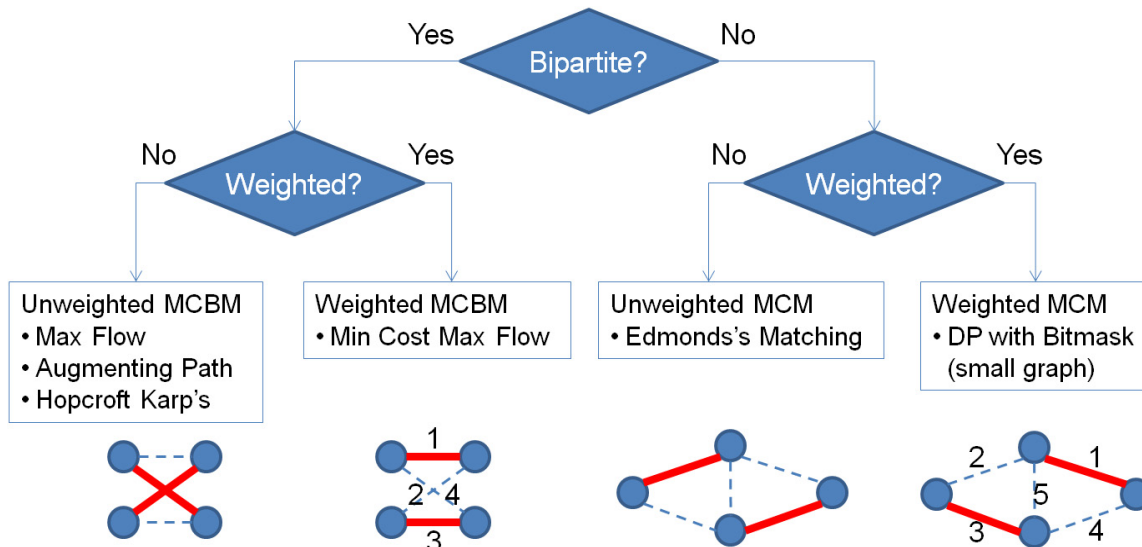
Figure 9.4: The Four Common Variants of Graph Matching in Programming Contests

---

[9]There are other Graph Matching variants outside these four, e.g. the Stable Marriage problem. However, we will concentrate on these four variants in this section.

Solution(s)

Solutions for Unweighted MCBM

This variant is the easiest and several solutions have been discussed earlier in Section 4.6 (Network Flow) and Section 4.7.4 (Bipartite Graph). The list below summarizes three possible solutions for the Unweighted MCBM problems:

1. Reducing the Unweighted MCBM problem into a Max Flow Problem.
   See Section 4.6 and 4.7.4 for details.
   The time complexity depends on the chosen Max Flow algorithm.

2. $O(V^2 + VE)$ Augmenting Path Algorithm for Unweighted MCBM.
   See Section 4.7.4 for details.
   This is good enough for various contest problems involving Unweighted MCBM.

3. $O(\sqrt{V}E)$ Hopcroft Karp's Algorithm for Unweighted MCBM
   See Section 9.12 for details.

Solutions for Weighted MCBM

When the edges in the bipartite graph are weighted, not all possible MCBMs are optimal. We need to pick one (not necessarily unique) MCBM that has the minimum overall total weight. One possible solution[10] is to reduce the Weighted MCBM problem into a Min Cost Max Flow (MCMF) problem (see Section 9.23).

For example, in Figure 9.5, we show one test case of UVa 10746 - Crime Wave - The Sequel. This is an MCBM problem on a complete bipartite graph $K_{n,m}$, but each edge has associated cost. We add edges from source s to vertices of the left set with capacity 1 and cost 0. We also add edges from vertices of the right set to the sink t also with capacity 1 and cost 0. The directed edges from the left set to the right set has capacity 1 and cost according to the problem description. After having this weighted flow graph, we can run the MCMF algorithm as shown in Section 9.23 to get the required answer: Flow $1 = 0 \to 2 \to 4 \to 8$ with cost 5, Flow $2 = 0 \to 1 \to 4 \to 2$ (cancel flow 2-4) $\to 6 \to 8$ with cost 15, and Flow 3 $= 0 \to 3 \to 5 \to 8$ with cost 20. The minimum total cost is $5 + 15 + 20 = 40$.
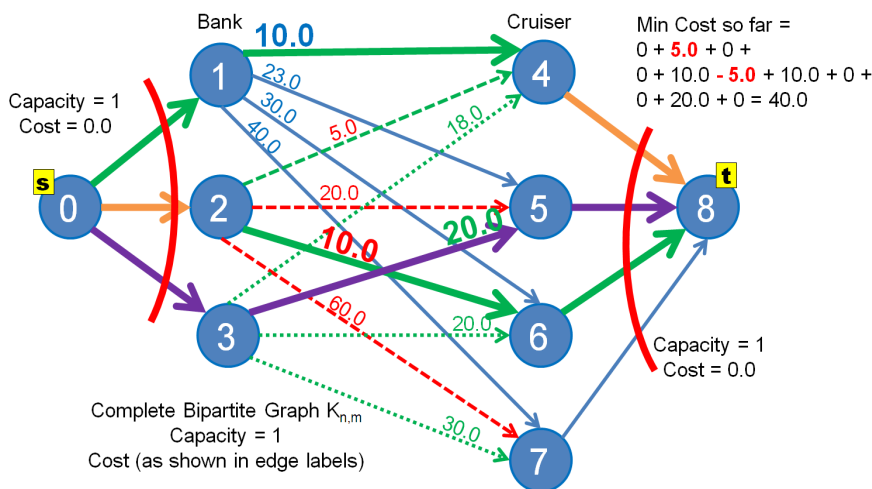
Figure 9.5: A Sample Test Case of UVa 10746: 3 Matchings with Min Cost $= 40$

---

[10]Another possible solution if we want to get *perfect* bipartite matching with minimum cost is the Hungarian (Kuhn-Munkres's) algorithm.

Solutions for Unweighted MCM

While the graph matching problem is easy on bipartite graphs, it is hard on general graphs. In the past, computer scientists thought that this variant was another NP-Complete problem until Jack Edmonds published a polynomial algorithm for solving this problem in his 1965 paper titled "Paths, trees, and flowers" [13].

The main issue is that on general graph, we may encounter odd-length augmenting cycles. Edmonds calls such a cycle a 'blossom'. The key idea of Edmonds Matching algorithm is to repeatedly shrink these blossoms (potentially in recursive fashion) so that finding augmenting paths returns back to the easy case as in bipartite graph. Then, Edmonds matching algorithm readjust the matchings when these blossoms are re-expanded (lifted).

The implementation of Edmonds Matching algorithm is not straightforward. Therefore, to make this graph matching variant more manageable, many problem authors limit the size of their unweighted general graphs to be small enough, i.e. $V \leq 18$ so that an $O(V \times 2^V)$ DP with bitmask algorithm can be used to solve it (see **Exercise 8.3.1.1**).

Solution for Weighted MCM

This is potentially the hardest variant. The given graph is a general graph and the edges have associated weights. In typical programming contest environment, the most likely solution is the DP with bitmask (see Section 8.3.1) as the problem authors usually set the problem on *a small general graph* only.

Visualization of Graph Matching

To help readers in understanding these graph matching variants and their solutions, we have built the following visualization tool:

Visualization: `www.comp.nus.edu.sg/~stevenha/visualization/matching.html`

In this visualization tool, you can draw your own graph and the system will present the correct graph matching algorithm(s) based on the two characteristics: Whether the input graph is bipartite and/or weighted. Note that the visualization of Edmonds's Matching inside our tool is probably one of the first in the world.

Exercise 9.10.1*: Implement Kuhn Munkres's algorithm! (see the original paper [39, 45]).

Exercise 9.10.2*: Implement Edmonds's Matching algorithm! (see the original paper [13]).

Programming exercises related to Graph Matching:

- See some assignment problems (bipartite matching with capacity) in Section 4.6
- See some Unweighted MCBM problems and variants in Section 4.7.4
- See some Weighted MCBM problems in Section 9.23
- Unweighted MCM
  1. **UVa 11439 - Maximizing the ICPC *** (binary search the answer to get the minimum weight; use this weight to reconstruct the graph; use Edmonds's Matching algorithm to test if we can get perfect matching on general graph)
- See (Un)weighted MCM problems on *small general graph* in Section 8.3 (DP)

9.11 Great-Circle Distance

Problem Description

Sphere is a perfectly round geometrical object in 3D space.

The **Great-Circle Distance** between any two points A and B on sphere is the shortest distance along a path on the **surface of the sphere**. This path is an *arc* of the **Great-Circle** of that sphere that pass through the two points A and B. We can imagine Great-Circle as the resulting circle that appears if we cut the sphere with a plane so that we have two *equal* hemispheres (see Figure 9.6—left and middle).
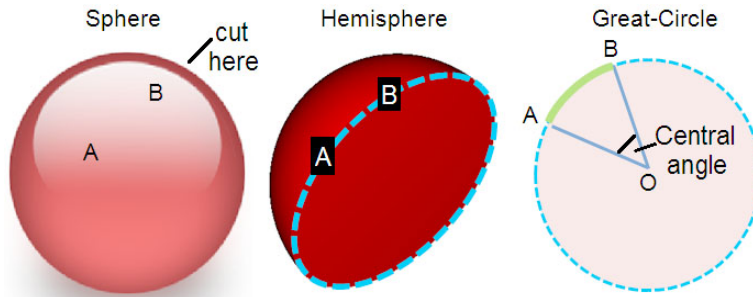
Figure 9.6: L: Sphere, M: Hemisphere and Great-Circle, R: gcDistance (Arc A-B)

Solution(s)

To find the Great-Circle Distance, we have to find the central angle AOB (see Figure 9.6—right) of the Great-Circle where O is the center of the Great-Circle (which is also the center of the sphere). Given the radius of the sphere/Great-Circle, we can then determine the length of arc A-B, which is the required Great-Circle distance.

Although quite rare nowadays, some contest problems involving 'Earth', 'Airlines', etc use this distance measurement. Usually, the two points on the surface of a sphere are given as the Earth coordinates, i.e. the (latitude, longitude) pair. The following library code will help us to obtain the Great-Circle distance given two points on the sphere and the radius of the sphere. We omit the derivation as it is not important for competitive programming.

```
double gcDistance(double pLat, double pLong,
                  double qLat, double qLong, double radius) {
  pLat *= PI / 180; pLong *= PI / 180;    // convert degree to radian
  qLat *= PI / 180; qLong *= PI / 180;
  return radius * acos(cos(pLat)*cos(pLong)*cos(qLat)*cos(qLong) +
                       cos(pLat)*sin(pLong)*cos(qLat)*sin(qLong) +
                       sin(pLat)*sin(qLat)); }
```

Source code: UVa11817.cpp/java

Programming exercises related to Great-Circle Distance:

1. UVa 00535 - Globetrotter (`gcDistance`)
2. UVa 10316 - Airline Hub (`gcDistance`)
3. UVa 10897 - Travelling Distance (`gcDistance`)
4. UVa 11817 - Tunnelling The Earth (`gcDistance`; 3D Euclidean distance)

9.12 Hopcroft Karp's Algorithm

Hopcroft Karp's algorithm [28] is another algorithm to solve the Unweighted Maximum Cardinality Bipartite Matching (MCBM) problem on top of the Max Flow based solution (which is longer to code) and the Augmenting Path algorithm (which is the preferred method) as discussed in Section 4.7.4.

In our opinion, the main reason for using the longer-to-code Hopcroft Karp's algorithm instead of the simpler-and-shorter-to-code Augmenting Path algorithm to solve the Unweighted MCBM is its runtime speed. Hopcroft Karp's algorithm runs in $O(\sqrt{V}E)$ which is (much) faster than the $O(VE)$ Augmenting Path algorithm on medium-sized ($V \approx 500$) bipartite (and dense) graphs.

An extreme example is a Complete Bipartite Graph $K_{n,m}$ with $V = n+m$ and $E = n \times m$. On such bipartite graph, the Augmenting Path algorithm has worst case time complexity of $O((n + m) \times n \times m)$. If $m = n$, we have an $O(n^3)$ solution which is only OK for $n \leq 200$.

The main issue with the $O(VE)$ Augmenting Path algorithm is that it may explore the longer augmenting paths first (as it is essentially a 'modified DFS'). This is not efficient. By exploring the *shorter* augmenting paths first, Hopcroft and Karp proved that their algorithm will only run in $O(\sqrt{V})$ iterations [28]. In each iteration, Hopcroft Karp's algorithm executes an $O(E)$ BFS from all the free vertices on the left set and finds augmenting paths of increasing lengths (starting from length 1: a free edge, length 3: a free edge, a matched edge, and a free edge again, length 5, length 7, and so on...). Then, it calls another $O(E)$ DFS to augment those augmenting paths (Hopcroft Karp's algorithm can increase *more than one matching* in one algorithm iteration). Therefore, the overall time complexity of Hopcroft Karp's algorithm is $O(\sqrt{V}E)$.

For the extreme example on Complete Bipartite Graph $K_{n,m}$ shown above, the Hopcroft Karp's algorithm has worst case time complexity of $O(\sqrt{(n + m)} \times n \times m)$. If $m = n$, we have an $O(n^{\frac{5}{2}})$ solution which is OK for $n \leq 600$. Therefore, if the problem author is 'nasty enough' to set $n \approx 500$ and relatively dense bipartite graph for an Unweighted MCBM problem, using Hopcroft Karp's is safer than the standard Augmenting Path algorithm (however, see **Exercise 4.7.4.3*** for a trick to make Augmenting Path algorithm runs 'fast enough' even if the input is a relatively dense and large bipartite graph).

Exercise 9.12.1*: Implement the Hopcroft Karp's algorithm starting from the Augmenting Path algorithm shown in Section 4.7.4 using the idea shown above.

Exercise 9.12.2*: Investigate the similarities and differences of Hopcroft Karp's algorithm and Dinic's algorithm shown in Section 9.7!

9.13   Independent and Edge-Disjoint Paths

Problem Description

Two paths that start from a source vertex s to a sink vertex t are said to be *independent* (vertex-disjoint) if they do not share any vertex apart from s and t.

Two paths that start from a source s to sink t are said to be edge-disjoint if they do not share any edge (but they can share vertices other than s and t).

Given a graph G, find the maximum number of independent and edge-disjoint paths from source s to sink t.

Solution(s)

The problem of finding the (maximum number of) independent paths from source s to sink t can be reduced to the Network (Max) Flow problem. We construct a flow network $N = (V, E)$ from G with vertex capacities, where N is the carbon copy of G except that the capacity of each $v \in V$ is 1 (i.e. each vertex can only be used once—see how to deal with vertex capacity in Section 4.6) and the capacity of each $e \in E$ is also 1 (i.e. each edge can only be used once too). Then run the Edmonds Karp's algorithm as per normal.
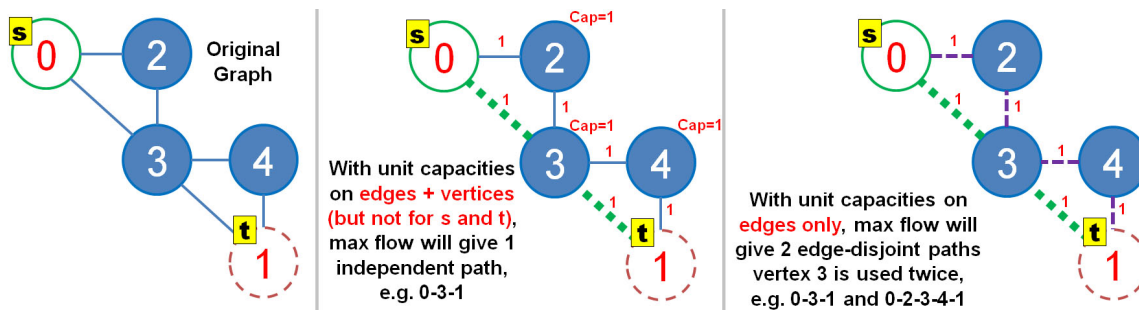
Figure 9.7: Comparison Between Max Independent Paths vs Max Edge-Disjoint Paths

Finding the (maximum number of) edge-disjoint paths from s to t is similar to finding (maximum) independent paths. The only difference is that this time we do not have any vertex capacity which implies that two edge-disjoint paths can still share the same vertex. See Figure 9.7 for a comparison between maximum independent paths and edge-disjoint paths from $s = 0$ to $t = 6$.

Programming exercises related to Independent and Edge-Disjoint Paths problem:

1. **UVa 00563 - Crimewave *** (check whether the maximum number of independent paths on the flow graph—with unit edge and unit vertex capacity—equals to b banks; analyze the upperbound of the answer to realize that the standard max flow solution suffices even for the largest test case)

2. **UVa 01242 - Necklace *** (LA 4271, Hefei08, to have a necklace, we need to be able to *two* edge-disjoint s-t flows)

9.14 Inversion Index

Problem Description

Inversion index problem is defined as follows: Given a list of numbers, count the minimum number of 'bubble sort' swaps (swap between pair of consecutive items) that are needed to make the list sorted in (usually ascending) order.

For example, if the content of the list is {3, 2, 1, 4}, we need 3 'bubble sort' swaps to make this list sorted in ascending order, i.e. swap (3, 2) to get {2, 3, 1, 4}, swap (3, 1) to get {2, 1, 3, 4}, and finally swap (2, 1) to get {1, 2, 3, 4}.

Solution(s)

$O(n^2)$ solution

The most obvious solution is to count how many swaps are needed during the actual running of the $O(n^2)$ bubble sort algorithm.

$O(n \log n)$ solution

The better $O(n \log n)$ Divide and Conquer solution for this inversion index problem is to modify merge sort. During the merge process of merge sort, if the front of the right sorted sublist is taken first rather than the front of the left sorted sublist, we say that 'inversion occurs'. Add inversion index counter by the size of the current left sublist. When merge sort is completed, report the value of this counter. As we only add $O(1)$ steps to merge sort, this solution has the same time complexity as merge sort, i.e. $O(n \log n)$.

On the example above, we initially have: {3, 2, 1, 4}. Merge sort will split this into sublist {3, 2} and {1, 4}. The left sublist will cause one inversion as we have to swap 3 and 2 to get {2, 3}. The right sublist {1, 4} will not cause any inversion as it is already sorted. Now, we merge {2, 3} with {1, 4}. The first number to be taken is 1 from the front of the right sublist. We have two more inversions because the left sublist has two members: {2, 3} that have to be swapped with 1. There is no more inversion after this. Therefore, there are a total of 3 inversions for this example.

Programming exercises related to Inversion Index problem:

1. UVa 00299 - Train Swapping (solvable with $O(n^2)$ bubble sort)
2. **UVa 00612 - DNA Sorting *** (needs $O(n^2)$ `stable_sort`)
3. **UVa 10327 - Flip Sort *** (solvable with $O(n^2)$ bubble sort)
4. UVa 10810 - Ultra Quicksort (requires $O(n \log n)$ merge sort)
5. UVa 11495 - Bubbles and Buckets (requires $O(n \log n)$ merge sort)
6. **UVa 11858 - Frosh Week *** (requires $O(n \log n)$ merge sort; 64-bit integer)

9.15 Josephus Problem

Problem Description

The Josephus problem is a classic problem where initially there are n people numbered from 1, 2, ..., n, standing in a circle. Every k-th person is going to be executed and removed from the circle. This count-then-execute process is repeated until there is only one person left and this person will be saved (history said that he was the person named Josephus).

Solution(s)

Complete Search for Smaller Instances

The smaller instances of Josephus problem are solvable with Complete Search (see Section 3.2) by simply simulating the process with help of a cyclic array (or a circular linked list). The larger instances of Josephus problem require better solutions.

Special Case when $k = 2$

There is an elegant way to determine the position of the last surviving person for $k = 2$ using binary representation of the number n. If $n = 1b_1b_2b_3..b_n$ then the answer is $b_1b_2b_3..b_n1$, i.e. we move the most significant bit of n to the back to make it the least significant bit. This way, the Josephus problem with $k = 2$ can be solved in $O(1)$.

General Case

Let $F(n, k)$ denotes the position of the survivor for a circle of size n and with k skipping rule and we number the people from 0, 1, ..., $n - 1$ (we will later add $+1$ to the final answer to match the format of the original problem description above). After the k-th person is killed, the circle shrinks by one to size $n - 1$ and the position of the survivor is now $F(n - 1, k)$. This relation is captured with equation $F(n, k) = (F(n - 1, k) + k)\%n$. The base case is when $n = 1$ where we have $F(1, k) = 0$. This recurrence has a time complexity of $O(n)$.

Other Variants

Josephus problem has several other variants that cannot be name one by one in this book.

Programming exercises related to Josephus problem:

1. UVa 00130 - Roman Roulette (the original Josephus problem)
2. UVa 00133 - The Dole Queue (brute force, similar to UVa 130)
3. UVa 00151 - Power Crisis (the original Josephus problem)
4. UVa 00305 - Joseph (the answer can be precalculated)
5. UVa 00402 - M*A*S*H (modified Josephus, simulation)
6. UVa 00440 - Eeny Meeny Moo (brute force, similar to UVa 151)
7. UVa 10015 - Joseph's Cousin (modified Josephus, dynamic k, variant of UVa 305)
8. ***UVa 10771 - Barbarian tribes*** * (brute force, input size is small)
9. ***UVa 10774 - Repeated Josephus*** * (repeated case of Josephus when $k = 2$)
10. ***UVa 11351 - Last Man Standing*** * (use general case Josephus recurrence)

9.16   Knight Moves

Problem Description

In chess, a knight can move in an interesting 'L-shaped' way. Formally, a knight can move from a cell (r_1, c_1) to another cell (r_2, c_2) in an $n \times n$ chessboard if and only if $(r_1 - r_2)^2 + (c_1 - c_2)^2 = 5$. A common query is the length of shortest moves to move a knight from a starting cell to another target cell. There can be many queries on the same chessboard.

Solution(s)

One BFS per Query

If the chessboard size is small, we can afford to run one BFS per query. For each query, we run BFS from the starting cell. Each cell has at most 8 edges connected to another cells (some cells around the border of the chessboard have less edges). We stop BFS as soon as we reach the target cell. We can use BFS on this shortest path problem as the graph is unweighted (see Section 4.4.2). As there are up to $O(n^2)$ cells in the chessboard, the overall time complexity is $O(n^2 + 8n^2) = O(n^2)$ per query or $O(Qn^2)$ if there are Q queries.

One Precalculated BFS and Handling Special Cases

The solution above is not the most efficient way to solve this problem. If the given chessboard is large and there are several queries, e.g. $n = 1000$ and $Q = 16$ in UVa 11643 - Knight Tour, the approach above will get TLE.

A better solution is to realize that if the chessboard is large enough and we pick two random cells (r_a, c_a) and (r_b, c_b) in the middle of the chessboard with shortest knight moves of d steps between them, shifting the cell positions by a constant factor does not change the answer, i.e. the shortest knight moves from $(r_a + k, c_a + k)$ and $(r_b + k, c_b + k)$ is also d steps, for a constant factor k.

Therefore, we can just run *one* BFS from an arbitrary source cell and do some adjustments to the answer. However, there are a few special (literally) corner cases to be handled. Finding these special cases can be a headache and many Wrong Answers are expected if one does not know them yet. To make this section interesting, we purposely leave this crucial last step as a starred exercise. Try solving UVa 11643 after you get these answers.

Exercise 9.16.1*: Find those special cases and address them. Hints:

1. Separate cases when $3 \leq n \leq 4$ and $n \geq 5$.

2. Literally concentrate on corner cells and side cells.

3. What happen if the starting cell and the target cell are too close?

Programming exercises related to Knight Tour problem:

1. **UVa 00439 - Knight Moves *** (one BFS per query is enough)

2. *__UVa 11643 - Knight Tour *__* (the distance between any 2 interesting positions can be obtained by using a pre-calculated BFS table (plus handling of the special corner cases); afterwards, this is just classic TSP problem, see Section 3.5.2)

9.17 Kosaraju's Algorithm

Finding Strongly Connected Components (SCCs) of a directed graph is a classic graph problem that has been discussed in Section 4.2.9. We have seen a modified DFS called Tarjan's algorithm that can solve this problem in efficient $O(V + E)$ time.

In this section, we present another DFS-based algorithm that can be used to find SCCs of a directed graph called the Kosaraju's algorithm. The basic idea of this algorithm is to run DFS *twice*. The *first* DFS is done on the *original directed graph* and record the 'post-order' traversal of the vertices as in finding topological sort[11] in Section 4.2.5. The *second* DFS is done on the *transpose of the original directed graph* using the 'post-order' ordering found by the first DFS. This two passes of DFS is enough to find the SCCs of the directed graph. The C++ implementation of this algorithm is shown below. We encourage readers to read more details on how this algorithm works from another source, e.g. [7].

```
void Kosaraju(int u, int pass) {      // pass = 1 (original), 2 (transpose)
  dfs_num[u] = 1;
  vii neighbor;             // use different Adjacency List in the two passes
  if (pass == 1) neighbor = AdjList[u]; else neighbor = AdjListT[u];
  for (int j = 0; j < (int)neighbor.size(); j++) {
    ii v = neighbor[j];
    if (dfs_num[v.first] == DFS_WHITE)
      Kosaraju(v.first, pass);
  }
  S.push_back(u);        // as in finding topological order in Section 4.2.5
}


// in int main()
  S.clear();  // first pass is to record the 'post-order' of original graph
  dfs_num.assign(N, DFS_WHITE);
  for (i = 0; i < N; i++)
    if (dfs_num[i] == DFS_WHITE)
      Kosaraju(i, 1);
  numSCC = 0;   // second pass: explore the SCCs based on first pass result
  dfs_num.assign(N, DFS_WHITE);
  for (i = N - 1; i >= 0; i--)
    if (dfs_num[S[i]] == DFS_WHITE) {
      numSCC++;
      Kosaraju(S[i], 2); // AdjListT -> the transpose of the original graph
    }

  printf("There are %d SCCs\n", numSCC);
```

Source code: `UVa11838.cpp/java`

Kosaraju's algorithm requires graph transpose routine (or build two graph data structures upfront) that is mentioned briefly in Section 2.4.1 and it needs two passes through the graph data structure. Tarjan's algorithm presented in Section 4.2.9 does not need graph transpose routine and it only needs only one pass. However, these two SCC finding algorithms are equally good and can be used to solve many (if not all) SCC problems listed in this book.

[11]But this may not be a valid topological sort as the input directed graph may be cyclic.

9.18 Lowest Common Ancestor

Problem Description

Given a rooted tree T with n vertices, the Lowest Common Ancestor (LCA) between two vertices u and v, or $LCA(u, v)$, is defined as the lowest vertex in T that has both u and v as descendants. We allow a vertex to be a descendant of itself, i.e. there is a possibility that $LCA(u, v) = u$ or $LCA(u, v) = v$.
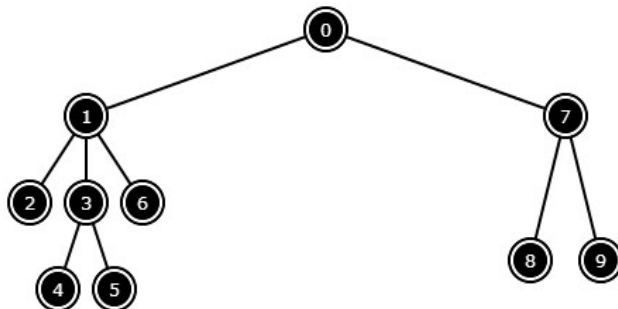
Figure 9.8: An example of a rooted tree T with $n = 10$ vertices

For example, in Figure 9.8, verify that the $LCA(4, 5) = 3$, $LCA(4, 6) = 1$, $LCA(4, 1) = 1$, $LCA(8, 9) = 7$, $LCA(4, 8) = 0$, and $LCA(0, 0) = 0$.

Solution(s)

Complete Search Solution

A naïve solution is to do two steps: From the first vertex u, we go all the way up to the root of T and record all vertices traversed along the way (this can be $O(n)$ if the tree is a very unbalanced). From the second vertex v, we also go all the way up to the root of T, but this time we stop if we encounter a common vertex for the first time (this can also be $O(n)$ if the $LCA(u, v)$ is the root and the tree is very unbalanced). This common vertex is the LCA. This requires $O(n)$ per (u, v) query and can be very slow if there are many queries.

For example, if we want to compute $LCA(4, 6)$ of the tree in Figure 9.8 using this complete search solution, we will first traverse path $4 \to 3 \to 1 \to 0$ and record these 4 vertices. Then, we traverse path $6 \to 1$ and then stop. We report that the LCA is vertex 1.

Reduction to Range Minimum Query

We can reduce the LCA problem into a Range Minimum Query (RMQ) problem (see Section 2.4.3). If the structure of the tree T is not changed throughout all Q queries, we can use the Sparse Table data structure with $O(n \log n)$ construction time and $O(1)$ RMQ time. The details on the Sparse Table data structure is shown in Section 9.33. In this section, we highlight the reduction process from LCA to RMQ.

We can reduce LCA to RMQ in linear time. The key idea is to observe that $LCA(u, v)$ is the shallowest vertex in the tree that is visited between the visits of u and v during a DFS traversal. So what we need to do is to run a DFS on the tree and record information about the depth and the time of visit for every node. Notice that we will visit a total of $2 * n - 1$ vertices in the DFS since the internal vertices will be visited several times. We need to build three arrays during this DFS: `E[0..2*n-2]` (which records the sequence of visited nodes), `L[0..2*n-2]` (which records the depth of each visited node), and `H[0..N-1]` (where `H[i]` records the index of the first occurrence of node `i` in `E`).

The key portion of the implementation is shown below:

```
int L[2*MAX_N], E[2*MAX_N], H[MAX_N], idx;

void dfs(int cur, int depth) {
  H[cur] = idx;
  E[idx] = cur;
  L[idx++] = depth;
  for (int i = 0; i < children[cur].size(); i++) {
    dfs(children[cur][i], depth+1);
    E[idx] = cur;                              // backtrack to current node
    L[idx++] = depth;
  }
}

void buildRMQ() {
  idx = 0;
  memset(H, -1, sizeof H);
  dfs(0, 0);                           // we assume that the root is at index 0
}
```

Source code: `LCA.cpp/java`

For example, if we call `dfs(0, 0)` on the tree in Figure 9.8, we will have[12]:

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
H	0	1	2	4	5	7	10	13	14	16	-1	-1	-1	-1	-1	-1	-1	-1	-1
E	0	1	2	1	3	4	3	5	3	*(1)*	6	1	0	7	8	7	9	7	0
L	0	1	2	1	2	<u>3</u>	<u>2</u>	<u>3</u>	<u>2</u>	<u>1</u>	<u>2</u>	1	0	1	2	1	2	1	0

Table 9.1: The Reduction from LCA to RMQ

Once we have these three arrays to work with, we can solve LCA using RMQ. Assume that `H[u]` $<$ `H[v]` or swap u and v otherwise. We notice that the problem reduces to finding the vertex with the smallest depth in `E[H[u]..H[v]]`. So the solution is given by $LCA(u,v)$ = `E[RMQ(H[u], H[v])]` where `RMQ(i, j)` is executed on the `L` array. If using the Sparse Table data structure shown in Section 9.33, it is the `L` array that needs to be processed in the construction phase.

For example, if we want to compute $LCA(4,6)$ of the tree in Figure 9.8, we will compute `H[4]` = 5 and `H[6]` = 10 and find the vertex with the smallest depth in `E[5..10]`. Calling `RMQ(5, 10)` on array `L` (see the underlined entries in row `L` of Table 9.1) returns index 9. The value of `E[9]` $= 1$ (see the italicized entry in row `E` of Table 9.1), therefore we report 1 as the answer of $LCA(4,6)$.

Programming exercises related to LCA:

1. UVa 10938 - Flea circus (Lowest Common Ancestor as outlined above)

2. *UVa 12238 - Ants Colony* (very similar to UVa 10938)

---

[12]In Section 4.2.1, `H` is named as `dfs_num`.

9.19   Magic Square Construction (Odd Size)

Problem Description

A magic square is a 2D array of size $n \times n$ that contains integers from $[1..n^2]$ with 'magic' property: The sum of integers in each row, column, and diagonal is the same. For example, for $n = 5$, we can have the following magic square below that has row sums, column sums, and diagonal sums equals to 65.

$$\begin{bmatrix} 17 & 24 & 1 & 8 & 15 \\ 23 & 5 & 7 & 14 & 16 \\ 4 & 6 & 13 & 20 & 22 \\ 10 & 12 & 19 & 21 & 3 \\ 11 & 18 & 25 & 2 & 9 \end{bmatrix}$$

Our task is to construct a magic square given its size n, assuming that n is odd.

Solution(s)

If we do not know the solution, we may have to use the standard recursive backtracking routine that try to place each integer $\in [1..n^2]$ one by one. Such Complete Search solution is too slow for large n.

Fortunately, there is a nice 'construction strategy' for magic square of odd size called the 'Siamese (De la Loubère) method'. We start from an empty 2D square array. Initially, we put integer 1 in the middle of the first row. Then we move northeast, wrapping around as necessary. If the new cell is currently empty, we add the next integer in that cell. If the cell has been occupied, we move one row down and continue going northeast. This Siamese method is shown in Figure 9.9. We reckon that deriving this strategy without prior exposure to this problem is likely not straightforward (although not impossible if one stares at the structure of several odd-sized Magic Squares long enough).
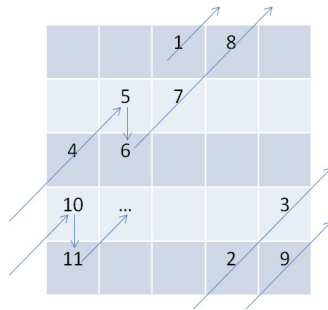
Figure 9.9: The Magic Square Construction Strategy for Odd n

There are other special cases for Magic Square construction of different sizes. It may be unnecessary to learn all of them as most likely it will not appear in programming contest. However, we can imagine some contestants who know such Magic Square construction strategies will have advantage in case such problem appears.

Programming exercises related to Magic Square:

1. **UVa 01266 - Magic Square** * (follow the given construction strategy)

9.20 Matrix Chain Multiplication

Problem Description

Given n matrices: $A_1, A_2, \ldots, A_n$, each A_i has size $P_{i-1} \times P_i$, output a complete parenthesized product $A_1 \times A_2 \times \ldots \times A_n$ that minimizes the number of scalar multiplications. A product of matrices is called completely parenthesized if it is either:

1. A single matrix

2. The product of 2 completely parenthesized products surrounded by parentheses

For example, given 3 matrices array $P = \{10, 100, 5, 50\}$ (which implies that matrix A_1 has size 10×100, matrix A_2 has size 100×5, and matrix A_3 has size 5×50. We can completely parenthesize these three matrices in two ways:

1. $(A_1 \times (A_2 \times A_3)) = 100 \times 5 \times 50 + 10 \times 100 \times 50 = 75000$ scalar multiplications

2. $((A_1 \times A_2) \times A_3) = 10 \times 100 \times 5 + 10 \times 5 \times 50 = 7500$ scalar multiplications

From the example above, we can see that the cost of multiplying these 3 matrices—in terms of the number of scalar multiplications—depends on the choice of the complete parenthesization of the matrices. However, exhaustively checking all possible complete parenthesizations is too slow as there are a huge number of such possibilities (for interested reader, there are $Cat(n-1)$ complete parenthesization of n matrices—see Section 5.4.3).

Matrix Multiplication

We can multiple two matrices a of size $p \times q$ and b of size $q \times r$ if the number of columns of a is the same as the number of rows of b (the inner dimension agree). The result of this multiplication is matrix c of size $p \times r$. The cost of such valid matrix multiplication is $O(p \times q \times r)$ multiplications and can be implemented with a short C++ code as follows:

```cpp
#define MAX_N 10                          // increase/decrease this value as needed
struct Matrix { int mat[MAX_N][MAX_N]; };

Matrix matMul(Matrix a, Matrix b, int p, int q, int r) {          // O(pqr)
  Matrix c; int i, j, k;
  for (i = 0; i < p; i++)
    for (j = 0; j < r; j++)
      for (c.mat[i][j] = k = 0; k < q; k++)
        c.mat[i][j] += a.mat[i][k] + b.mat[k][j];
  return c; }
```

For example, if we have 2×3 matrix a and 3×1 matrix b below, we need $2 \times 3 \times 1 = 6$ scalar multiplications.

$$\begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \end{bmatrix} \times \begin{bmatrix} b_{1,1} \\ b_{2,1} \\ b_{3,1} \end{bmatrix} = \begin{bmatrix} c_{1,1} = a_{1,1} \times b_{1,1} + a_{1,2} \times b_{2,1} + a_{1,3} \times b_{3,1} \\ c_{2,1} = a_{2,1} \times b_{1,1} + a_{2,2} \times b_{2,1} + a_{2,3} \times b_{3,1} \end{bmatrix}$$

When the two matrices are square matrices of size $n \times n$, this matrix multiplication runs in $O(n^3)$ (see Section 9.21 which is very similar with this one).

Solution(s)

This Matrix Chain Multiplication problem is usually one of the classic example to illustrate Dynamic Programming (DP) technique. As we have discussed DP in details in Section 3.5, we only outline the key ideas here. Note that for this problem, we do not actually multiply the matrices as shown in earlier subsection. We just need to find the optimal complete parenthesization of the n matrices.

Let $cost(i, j)$ where $i < j$ denotes the number of scalar multiplications needed to multiply matrix $A_i \times A_{i+1} \times \ldots \times A_j$. We have the following Complete Search recurrences:

1. $cost(i, j) = 0$ if $i = j$

2. $cost(i, j) = min(cost(i, k) + cost(k + 1, j) + P_{i-1} \times P_k \times P_j), \forall k \in [i \ldots j - 1]$

The optimal cost is stored in $cost(1, n)$. There are $O(n^2)$ different pairs of subproblem (i, j). Therefore, we need a DP table of size $O(n^2)$. Each subproblem requires up to $O(n)$ to be computed. Therefore, the time complexity of this DP solution for Matrix Chain Multiplication problem is $O(n^3)$.

Programming exercises related to Matrix Chain Multiplication:

1. **UVa 00348 - Optimal Array Mult ... *** (as above, output the optimal solution too; note that the optimal matrix multiplication sequence is not unique; e.g. imagine if all matrices are square matrices)

9.21 Matrix Power

Some Definitions and Sample Usages

In this section, we discuss a special case of matrix[13]: The *square matrix*[14]. To be precise, we discuss a special operation of square matrix: The *powers of a square matrix*. Mathematically, $M^0 = I$ and $M^p = \prod_{i=1}^{p} M$. I is the *Identity* matrix[15] and p is the given power of square matrix M. If we can do this operation in $O(n^3 \log p)$—which is the main topic of this subsection, we can solve some more interesting problems in programming contests, e.g.:

- Compute a *single*[16] Fibonacci number $fib(p)$ in $O(\log p)$ time instead of $O(p)$.
  Imagine if $p = 2^{30}$, $O(p)$ solution will get TLE but $\log_2(p)$ solution just need 30 steps.
  This is achievable by using the following equality:

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^p = \begin{bmatrix} fib(p+1) & \mathbf{fib(p)} \\ \mathbf{fib(p)} & fib(p-1) \end{bmatrix}$$

  For example, to compute $fib(11)$, we simply multiply the Fibonacci matrix 11 times, i.e. raise it to the power of 11. The answer is in the secondary diagonal of the matrix.

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{11} = \begin{bmatrix} 144 & \mathbf{89} \\ \mathbf{89} & 55 \end{bmatrix} = \begin{bmatrix} fib(12) & \mathbf{fib(11)} \\ \mathbf{fib(11)} & fib(10) \end{bmatrix}$$

- Compute the number of paths of length L of a graph stored in an Adjacency Matrix—which is a square matrix—in $O(n^3 \log L)$. Example: See the small graph of size $n = 4$ stored in an Adjacency Matrix M below. The various paths from vertex 0 to vertex 1 with different lengths are shown in entry $M[0][1]$ after M is raised to power L.

```
The graph:    0->1 with length 1: 0->1 (only 1 path)
              0->1 with length 2: impossible
  0--1        0->1 with length 3: 0->1->2->1 (and 0->1->0->1)
   |          0->1 with length 4: impossible
  2--3        0->1 with length 5: 0->1->2->3->2->1 (and 4 others)
```

$$M = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} M^2 = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 2 & 0 & 1 \\ 1 & 0 & 2 & 0 \\ 0 & 1 & 0 & 1 \end{bmatrix} M^3 = \begin{bmatrix} 0 & 2 & 0 & 1 \\ 2 & 0 & 3 & 0 \\ 0 & 3 & 0 & 2 \\ 1 & 0 & 2 & 0 \end{bmatrix} M^5 = \begin{bmatrix} 0 & 5 & 0 & 3 \\ 5 & 0 & 8 & 0 \\ 0 & 8 & 0 & 5 \\ 3 & 0 & 5 & 0 \end{bmatrix}$$

- Speed-up *some* DP problems as shown later in this section.

---

[13]A matrix is a rectangular (2D) array of numbers. Matrix of size $m \times n$ has m rows and n columns. The elements of the matrix is usually denoted by the matrix name with two subscripts.

[14]A square matrix is a matrix with the same number of rows and columns, i.e. it has size $n \times n$.

[15]Identity matrix is a matrix with all zeroes except that cells along the main diagonal are all ones.

[16]If we need $fib(n)$ **for all** $n \in$ [0..n], use $O(n)$ DP solution instead.

The Idea of Efficient Exponentiation (Power)

For the sake of discussion, let's assume that built-in library functions like *pow(base, p)* or other related functions that can raise a number *base* to a certain integer power *p* does not exist. Now, if we do exponentiation 'by definition' as shown below, we will have an inefficient $O(p)$ solution, especially if *p* is large[17].

```
int normalExp(int base, int p) {    // for simplicity, we use int data type
  int ans = 1;             // we also assume that ans will not exceed 2^31 - 1
  for (int i = 0; i < p; i++) ans *= base;                    // this is O(p)
  return ans; }
```

There is a better solution that uses Divide & Conquer principle. We can express A^p as:

$A^0 = 1$ (base case).
$A^1 = A$ (another base case, but see **Exercise 9.21.1**).
$A^p = A^{p-1} \times A$ if *p* is odd.
$A^p = (A^{p/2})^2$ if *p* is even.
As this approach keeps halving the value of *p* by two, it runs in $O(\log p)$.

For example, by definition: $2^9 = 2 \times 2 \times 2 \times 2 \times 2 \times 2 \times 2 \times 2 \times 2 \approx O(p)$ multiplications. But with Divide & Conquer: $2^9 = 2^8 \times 2 = (2^4)^2 \times 2 = ((2^2)^2)^2 \times 2 \approx O(\log p)$ multiplications.

A typical recursive implementation of this Divide & Conquer exponentiation—omitting cases when the answer exceeds the range of 32-bit integer—is shown below:

```
int fastExp(int base, int p) {                                    // O(log p)
      if (p == 0) return 1;
  else if (p == 1) return base;                    // See the Exercise below
  else {             int res = fastExp(base, p / 2); res *= res;
                     if (p % 2 == 1) res *= base;
                     return res; } }
```

Exercise 9.21.1*: Do we actually need the second base case: `if (p == 1) return base;`?

Exercise 9.21.2*: Raising a number to a certain (integer) power can easily cause overflow. An interesting variant is to compute $base^p (\bmod m)$. Rewrite function `fastExp(base, p)` into `modPow(base, p, m)` (also see Section 5.3.2 and Section 5.5.8)!

Exercise 9.21.3*: Rewrite the recursive implementation of Divide & Conquer implementation into an iterative implementation. Hint: Continue reading this section.

Square Matrix Exponentiation (Matrix Power)

We can use the same $O(\log p)$ efficient exponentiation technique shown above to perform square matrix exponentiation (matrix power) in $O(n^3 \log p)$ because each matrix multiplication[18] is $O(n^3)$. The *iterative* implementation (for comparison with the recursive implementation shown earlier) is shown below:

---

[17]If you encounter input size of 'gigantic' value in programming contest problems, like 1B, the problem author is *usually* looking for a logarithmic solution. Notice that $\log_2(1B) \approx \log_2(2^{30})$ is still just 30!

[18]There exists a faster but more complex algorithm for matrix multiplication: The $O(n^{2.8074})$ Strassen's algorithm. Usually we do not use this algorithm for programming contests. Multiplying two Fibonacci matrices shown in Section 9.21 only requires $2^3 = 8$ multiplications as $n = 2$. This can be treated as $O(1)$. Thus, we can compute $fib(p)$ in $O(\log p)$.

```
#define MAX_N 2 // Fibonacci matrix, increase/decrease this value as needed
struct Matrix { int mat[MAX_N][MAX_N]; };       // we will return a 2D array

Matrix matMul(Matrix a, Matrix b) {                               // O(n^3)
  Matrix ans; int i, j, k;
  for (i = 0; i < MAX_N; i++)
    for (j = 0; j < MAX_N; j++)
      for (ans.mat[i][j] = k = 0; k < MAX_N; k++)       // if necessary, use
        ans.mat[i][j] += a.mat[i][k] * b.mat[k][j];    // modulo arithmetic
  return ans; }

Matrix matPow(Matrix base, int p) {                        // O(n^3 log p)
  Matrix ans; int i, j;
  for (i = 0; i < MAX_N; i++) for (j = 0; j < MAX_N; j++)
    ans.mat[i][j] = (i == j);                        // prepare identity matrix
  while (p) {        // iterative version of Divide & Conquer exponentiation
    if (p & 1) ans = matMul(ans, base);     // if p is odd (last bit is on)
    base = matMul(base, base);                         // square the base
    p >>= 1;                                           // divide p by 2
  }
  return ans; }
```

Source code: UVa10229.cpp/java

DP Speed-up with Matrix Power

In this section, we discuss how to derive the required square matrices for two DP problems and show that raising these two square matrices to the required powers can speed-up the computation of the original DP problems.

We start with the 2×2 Fibonacci matrix. We know that $fib(0) = 0$, $fib(1) = 1$, and for $n \geq 2$, we have $fib(n) = fib(n-1) + fib(n-2)$. We can compute $fib(n)$ in $O(n)$ by using Dynamic Programming by computing $fib(n)$ *one by one* progressively from $[2..n]$. However, these DP transitions *can be made faster* by re-writing the Fibonacci recurrence into a matrix form as shown below:

First, we write two versions of Fibonacci recurrence as there are two terms in the recurrence:

$$fib(n+1) + fib(n) = fib(n+2)$$
$$fib(n) + fib(n-1) = fib(n+1)$$

Then, we re-write the recurrence into matrix form:

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \times \begin{bmatrix} fib(n+1) \\ fib(n) \end{bmatrix} = \begin{bmatrix} fib(n+2) \\ fib(n+1) \end{bmatrix}$$

Now we have $a \times fib(n+1) + b \times fib(n) = fib(n+2)$ and $c \times fib(n+1) + d \times fib(n) = fib(n+1)$. Notice that by writing the DP recurrence as shown above, we now have a 2×2 *square matrix*. The appropriate values for a, b, c, and d must be 1, 1, 1, 0 and this is the 2×2 Fibonacci matrix shown earlier. One matrix multiplication advances DP computation of Fibonacci number one step forward. If we multiply this 2×2 Fibonacci matrix p times, we advance DP computation of Fibonacci number p steps forward. We now have:

$$\underbrace{\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \times \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \times \ldots \times \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}}_{p} \times \begin{bmatrix} fib(n+1) \\ fib(n) \end{bmatrix} = \begin{bmatrix} fib(n+1+p) \\ fib(n+p) \end{bmatrix}$$

For example, if we set $n = 0$ and $p = 11$, and then use $O(\log p)$ matrix power instead of actually multiplying the matrix p times, we have the following calculations:

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{11} \times \begin{bmatrix} fib(1) \\ fib(0) \end{bmatrix} = \begin{bmatrix} 144 & 89 \\ 89 & 55 \end{bmatrix} \times \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 144 \\ \underline{\mathbf{89}} \end{bmatrix} = \begin{bmatrix} fib(12) \\ \underline{\mathbf{fib(11)}} \end{bmatrix}$$

This Fibonacci matrix can also be written as shown earlier, i.e.

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{p} = \begin{bmatrix} fib(p+1) & fib(p) \\ fib(p) & fib(p-1) \end{bmatrix}$$

Let's discuss one more example on how to derive the required square matrix for another DP problem: UVa 10655 - Contemplation! Algebra. The problem description is very simple: Given the value of $p = a + b$, $q = a \times b$, and n, find the value of $a^n + b^n$.

First, we tinker with the formula so that we can use $p = a + b$ and $q = a \times b$:

$$a^n + b^n = (a + b) \times (a^{n-1} + b^{n-1}) - (a \times b) \times (a^{n-2} + b^{n-2})$$

Next, we set $X_n = a^n + b^n$ to have $X_n = p \times X_{n-1} - q \times X_{n-2}$.
Then, we write this recurrence twice in the following form:

$$p \times X_{n+1} - q \times X_n = X_{n+2}$$
$$p \times X_n - q \times X_{n-1} = X_{n+1}$$

Then, we re-write the recurrence into matrix form:

$$\begin{bmatrix} p & -q \\ 1 & 0 \end{bmatrix} \times \begin{bmatrix} X_{n+1} \\ X_n \end{bmatrix} = \begin{bmatrix} X_{n+2} \\ X_{n+1} \end{bmatrix}$$

If we raise the 2×2 square matrix to the power of n (in $O(\log n)$ time) and then multiply the resulting square matrix with $X_1 = a^1 + b^1 = a + b = p$ and $X_0 = a^0 + b^0 = 1 + 1 = 2$, we have X_{n+1} and X_n. The required answer is X_n. This is faster than $O(n)$ standard DP computation for the same recurrence.

$$\begin{bmatrix} p & -q \\ 1 & 0 \end{bmatrix}^{n} \times \begin{bmatrix} X_1 \\ X_0 \end{bmatrix} = \begin{bmatrix} X_{n+1} \\ X_n \end{bmatrix}$$

Programming Exercises related to Matrix Power:

1. UVa 10229 - Modular Fibonacci (discussed in this section + modulo)
2. **UVa 10518 - How Many Calls? *** (derive the pattern of the answers for small n; the answer is $2 \times fib(n) - 1$; then use UVa 10229 solution)
3. **UVa 10655 - Contemplation, Algebra *** (discussed in this section)
4. UVa 10870 - Recurrences (form the required matrix first; power of matrix)
5. **UVa 11486 - Finding Paths in Grid *** (model as adjacency matrix; raise the adjacency matrix to the power of N in $O(\log N)$ to get the number of paths)
6. *UVa 12470 - Tribonacci* (very similar to UVa 10229; the 3×3 matrix is $= [0\ 1\ 0;$ $0\ 0\ 1;\ 1\ 1\ 1]$; the answer is at matrix[1][1] after it is raised to the power of n and with modulo 1000000009)

9.22 Max Weighted Independent Set

Problem Description

Given a *vertex-weighted* graph G, find the Max *Weighted* Independent Set (MWIS) of G. An Independent Set (IS)[19] is a set of vertices in a graph, no two of which are adjacent. Our task is to select an IS of G with the maximum total (vertex) weight. This is a hard problem on a general graph. However, if the given graph G is a tree or a bipartite graph, we have efficient solutions.

Solution(s)

On Tree

If graph G is a tree[20], we can find the MWIS of G using DP[21]. Let $C(v, taken)$ be the MWIS of the subtree rooted at v if it is *taken* as part of the MWIS. We have the following complete search recurrences:

1. If v is a leaf vertex

    (a) $C(v, true) = w(v)$
       % If leaf v is taken, then the weight of this subtree is the weight of this v.

    (b) $C(v, false) = 0$
       % If leaf v is not taken, then the weight of this subtree is 0.

2. If v is an internal vertex

    (a) $C(v, true) = w(v) + \sum_{\texttt{ch} \in \texttt{children(v)}} C(ch, false)$
       % If root v is taken, we add weight of v but all children of v *cannot* be taken.

    (b) $C(v, false) = \sum_{\texttt{ch} \in \texttt{children(v)}} max(C(ch, true), C(ch, false))$
       % If root v is not taken, children of v may or may not be taken.
       % We return the larger one.

The answer is $max(C(root, 1), C(root, 0))$—take or not take the root. This DP solution just requires $O(V)$ space and $O(V)$ time.

On Bipartite Graph

If the graph G is a bipartite graph, we have to reduce MWIS problem[22], into a Max Flow problem. We assign the original vertex cost (the weight of taking that vertex) as capacity from source to that vertex for the left set of the bipartite graph and capacity from that vertex to sink for right set of the bipartite graph. Then, we give 'infinite' capacity in between any edge in between the left and right sets. The MWIS of this bipartite graph is the weight of all vertex cost minus the max flow value of this flow graph.

---

[19]For your information, the complement of Independent Set is Vertex Cover.

[20]For most tree-related problems, we need to 'root the tree' first if it is not yet rooted. If the tree does not have a vertex dedicated as the root, pick an arbitrary vertex as the root. By doing this, the subproblems w.r.t subtrees may appear, like in this MWIS problem on Tree.

[21]Some optimization problems on *tree* may be solved with DP techniques. The solution usually involves passing information from/to parent and getting information from/to the children of a rooted tree.

[22]The non-weighted Max Independent Set (MIS) problem on bipartite graph can be reduced into a Max Cardinality Bipartite Matching (MCBM) problem—see Section 4.7.4.

9.23 Min Cost (Max) Flow

Problem Description

The Min Cost Flow problem is the problem of finding the *cheapest* possible way of sending a certain amount of (usually max) flow through a flow network. In this problem, every edge has two attributes: The flow capacity through this edge *and the unit cost* for sending one unit flow through this edge. Some problem authors choose to simplify this problem by setting the edge capacity to a constant integer and only vary the edge cost.
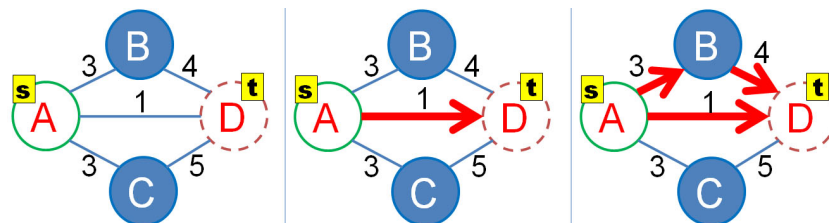
Figure 9.10: An Example of Min Cost Max Flow (MCMF) Problem (UVa 10594 [47])

Figure 9.10—left shows a (modified) instance of UVa 10594. Here, each edge has a uniform capacity of 10 units and a unit cost as shown in the edge label. We want to send 20 units of flow from A to D (note that the max flow of this flow graph is 30 units) which can be satisfied by sending 10 units of flow $A \rightarrow D$ with cost $1 \times 10 = 10$ (Figure 9.10—middle); plus another 10 units of flow $A \rightarrow B \rightarrow D$ with cost $(3 + 4) \times 10 = 70$ (Figure 9.10—right). The total cost is $10 + 70 = 80$ and this is the minimum. Note that if we choose to send the 20 units of flow via $A \rightarrow D$ (10 units) and $A \rightarrow \underline{C} \rightarrow D$ instead, we incur a cost of $1 \times 10 + (3 + \underline{5}) \times 10 = 10 + 80 = 90$. This is higher than the optimal cost of 80.

Solution(s)

The Min Cost (Max) Flow, or in short MCMF, can be solved by replacing the $O(E)$ BFS (to find the shortest—in terms of number of hops—augmenting path) in Edmonds Karp's algorithm into the $O(VE)$ Bellman Ford's (to find the shortest/cheapest—in terms of the *path cost*—augmenting path). We need a shortest path algorithm that can handle negative edge weights as such negative edge weights *may appear* when we cancel a certain flow along a backward edge (as we have to *subtract* the cost taken by this augmenting path as canceling flow means that we do not want to use that edge). See Figure 9.5 for an example.

The needs to use shortest path algorithm like Bellman Ford's slows down the MCMF implementation to around $O(V^2E^2)$ but this is usually compensated by the problem author of most MCMF problems by having smaller input graph constraints.

Programming exercises related to Min Cost (Max) Flow:

1. UVa 10594 - Data Flow (basic min cost max flow problem)
2. **UVa 10746 - Crime Wave - The Sequel** * (min *weighted* bip matching)
3. UVa 10806 - Dijkstra, Dijkstra (send 2 edge-disjoint flows with min cost)
4. ***UVa 10888 - Warehouse*** * (BFS/SSSP; min *weighted* bipartite matching)
5. ***UVa 11301 - Great Wall of China*** * (modeling, vertex capacity, MCMF)

9.24 Min Path Cover on DAG

Problem Description

The Min Path Cover (MPC) problem on DAG is described as the problem of finding the minimum number of paths to cover *each vertex* on DAG $G = (V, E)$. A path $v_0, v_1, \ldots, v_k$ is said to cover all vertices along its path.

Motivating problem—UVa 1201 - Taxi Cab Scheme: Imagine that the vertices in Figure 9.11.A are passengers, and we draw an edge between two vertices $u - v$ if one taxi can serve passenger u and then passenger v *on time*. The question is: What is the minimum number of taxis that must be deployed to serve *all* passengers?

The answer is two taxis. In Figure 9.11.D, we see one possible optimal solution. One taxi (dotted line) serves passenger 1, passenger 2, and then passenger 4. Another taxi (dashed line) serves passenger 3 and passenger 5. All passengers are served with just two taxis. Notice that there is one more optimal solution: $1 \to 3 \to 5$ and $2 \to 4$.
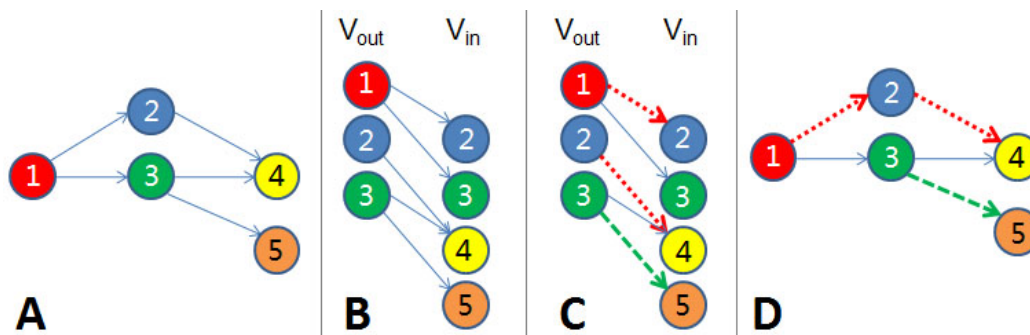
Figure 9.11: Min Path Cover on DAG (from UVa 1201 [47])

Solution(s)

This problem has a polynomial solution: Construct a *bipartite graph* $G' = (V_{out} \bigcup V_{in}, E')$ from G, where $V_{out} = \{v \in V : v \text{ has positive out-degree}\}$, $V_{in} = \{v \in V : v \text{ has positive in-degree}\}$, and $E' = \{(u, v) \in (Vout, Vin) : (u, v) \in E\}$. This G' is a bipartite graph. A matching on bipartite graph G' forces us to select at most one outgoing edge from every $u \in V_{out}$ (and similarly at most one incoming edge for $v \in V_{in}$). DAG G initially has n vertices, which can be covered with n paths of length 0 (the vertices themselves). One matching between vertex a and vertex b using edge (a, b) says that we can use one less path as edge $(a, b) \in E'$ can cover both vertices in $a \in V_{out}$ and $b \in V_{in}$. Thus if the MCBM in G' has size m, then we just need $n - m$ paths to cover each vertex in G.

The MCBM in G' that is needed to solve the MPC in G can be solved via several polynomial solutions, e.g. maximum flow solution, augmenting paths algorithm, or Hopcroft Karp's algorithm (see Section 9.10). As the solution for bipartite matching runs in polynomial time, the solution for the MPC in DAG also runs in polynomial time. Note that MPC in general graph is NP-hard.

Programming exercises related to Min Path Cover on DAG:

1. **UVa 01184 - Air Raid *** (LA 2696, Dhaka02, MPC on DAG ≈ MCBM)

2. **UVa 01201 - Taxi Cab Scheme *** (LA 3126, NWEurope04, MPC on DAG)

9.25 Pancake Sorting

Problem Description

Pancake Sorting is a classic[23] Computer Science problem, but it is rarely used. This problem can be described as follows: You are given a stack of **N** pancakes. The pancake at the bottom and at the top of the stack has **index 0** and **index N-1**, respectively. The size of a pancake is given by the pancake's diameter (**an integer** $\in$ [**1 .. MAX_D**]). All pancakes in the stack have **different** diameters. For example, a stack A of **N = 5** pancakes: {3, 8, 7, 6, 10} can be visualized as:

```
4 (top)         10
3                6
2                7
1                8
0 (bottom)       3
----------------------
index            A
```

Your task is to sort the stack in **descending order**—that is, the largest pancake is at the bottom and the smallest pancake is at the top. However, to make the problem more real-life like, sorting a stack of pancakes can only be done by a sequence of pancake 'flips', denoted by function **flip(i)**. A **flip(i)** move consists of inserting a spatula between two pancakes in a stack (at **index i** and **index N-1**) and flipping (reversing) the pancakes on the spatula (reversing the sub-stack [**i .. N-1**]).

For example, stack A can be transformed to stack B via **flip(0)**, i.e. inserting a spatula between index 0 and 4 then flipping the pancakes in between. Stack B can be transformed to stack C via **flip(3)**. Stack C can be transformed to stack D via **flip(1)**. And so on... Our target is to make the stack sorted in **descending order**, i.e. we want the final stack to be like stack E.

```
4 (top)     10 <--   3  <--   8  <--   6              3
3            6        8  <--   3        7       ...    6
2            7        7        7        3              7
1            8        6        6  <--   8              8
0 (bottom)   3  <--  10       10       10             10
--------------------------------------------------------------
index        A        B        C        D       ...    E
```

To make the task more challenging, you have to compute the **minimum number of flip(i) operations** that you need so that the stack of **N** pancakes is sorted in descending order.

You are given an integer T in the first line, and then T test cases, one in each line. Each test case starts with an integer N, followed by N integers that describe the initial content of the stack. You have to output one integer, the minimum number of **flip(i)** operations to sort the stack.

Constraints: **1 ≤ T ≤ 100, 1 ≤ N ≤ 10**, and **N ≤ MAX_D ≤ 1000000**.

---

[23]Bill Gates (Microsoft founder, former CEO, and current chairman) wrote only one research paper so far, and it is about this pancake sorting [22].

Sample Test Cases

Sample Input

```
7
4    4 3 2 1
8    8 7 6 5 4 1 2 3
5    5 1 2 4 3
5    555555 111111 222222 444444 333333
8    1000000 999999 999998 999997 999996 999995 999994 999993
5    3 8 7 6 10
10   8 1 9 2 0 5 7 3 6 4
```

Sample Output

```
0
1
2
2
0
4
11
```

Explanation

- The first stack is already sorted in descending order.

- The second stack can be sorted with one call of **flip(5)**.

- The third (and also the fourth) input stack can be sorted in descending order by calling **flip(3)** then **flip(1)**: 2 flips.

- The fifth input stack, although contains large integers, is already sorted in descending order, so 0 flip is needed.

- The sixth input stack is actually the sample stack shown in the problem description. This stack can be sorted in descending order using at minimum 4 flips, i.e. Solution 1: **flip(0)**, **flip(1)**, **flip(2)**, **flip(1)**: 4 flips. Solution 2: **flip(1)**, **flip(2)**, **flip(1)**, **flip(0)**: also 4 flips.

- The seventh stack with **N = 10** is for you to test the runtime speed of your solution.

Solution(s)

First, we need to make an observation that the diameters of the pancake do not really matter. We just need to write simple code to sort these (potentially huge) pancake diameters from [**1..1 million**] and relabel them to [**0..N-1**]. This way, we can describe any stack of pancakes as simply a permutation of N integers.

If we just need to get the pancakes sorted, we can use a non optimal $O(2 \times N - 3)$ Greedy algorithm: Flip the largest pancake to the top, then flip it to the bottom. Flip the second largest pancake to the top, then flip it to the second from bottom. And so on. If we keep doing this, we will be able to have a sorted pancake in $O(2 \times N - 3)$ steps, regardless of the initial state.

However, to get the minimum number of flip operations, we need to be able to model this problem as a Shortest Paths problem on unweighted State-Space graph (see Section 8.2.3). The vertex of this State-Space graph is a permutation of N pancakes. A vertex is connected with unweighted edges to $O(N-1)$ other vertices via various flip operations (minus one as flipping the topmost pancake does not change anything). We can then use BFS from the starting permutation to find the shortest path to the target permutation (where the permutation is sorted in descending order). There are up to $V = O(N!)$ vertices and up to $V = O(N! \times (N-1))$ in this State-Space graph. Therefore, an $O(V+E)$ BFS runs in $O(N \times N!)$ per test case or $O(T \times N \times N!)$ for all test cases. Note that coding such BFS is already a challenging task (see Section 4.4.2 and 8.2.3). But this solution is still too slow for the largest test case.

A simple optimization is to run BFS from the target permutation (sorted descending) to all other permutations **only once**, for all possible **N** in **[1..10]**. This solution has time complexity of roughly $O(10 \times N \times N! + T)$, much faster than before but still too slow for typical programming contest settings.

A better solution is a more sophisticated search technique called 'meet in the middle' (bidirectional BFS) to bring down the search space to a manageable level (see Section 8.2.4). First, we do some preliminary analysis (or we can also look at 'Pancake Number', http://oeis.org/A058986) to identify that for the largest test case when $N = 10$, we need *at most* 11 flips to sort any input stack to the sorted one. Therefore, we precalculate BFS from the target permutation to all other permutations for all **N** $\in$ **[1..10]**, but stopping as soon as we reach depth $\lfloor \frac{11}{2} \rfloor = 5$. Then, for each test case, we run BFS from the starting permutation again with maximum depth 5. If we encounter a common vertex with the precalculated BFS from target permutation, we know that the answer is the distance from starting permutation to this vertex plus the distance from target permutation to this vertex. If we do not encounter a common vertex at all, we know that the answer should be the maximum flips: 11. On the largest test case with $N = 10$ for all test cases, this solution has time complexity of roughly $O((10+T) \times 10^5)$, which is now feasible.

Programming exercises related to Pancake Sorting:

1. **UVa 00120 - Stacks Of Flapjacks *** (pancake sorting, greedy version)
2. The Pancake Sorting problem as described in this section.

9.26 Pollard's rho Integer Factoring Algorithm

In Section 5.5.4, we have seen the optimized trial division algorithm that can be used to find the prime factors of integers up to $\approx 9 \times 10^{13}$ (see **Exercise 5.5.4.1**) in *contest environment* (i.e. in 'a few seconds' instead of minutes/hours/days). Now, what if we are given a 64-bit unsigned integer (i.e. up to $\approx 1 \times 10^{19}$) to be factored in contest environment?

For a *faster* integer factorization, one can use the Pollard's rho algorithm [52, 3]. The key idea of this algorithm is that two integers x and y are congruent modulo p (p is one of the factor of n—the integer that we want to factor) with probability 0.5 after 'a few ($1.177\sqrt{p}$) integers' have been randomly chosen.

The theoretical details of this algorithm is probably not that important for Competitive Programming. In this section, we directly provide a working C++ implementation below which can be used to handle composite integer that fit in 64-bit unsigned integers in contest environment. However, Pollard's rho cannot factor an integer n if n is a large prime due to the way the algorithm works. To handle this case, we have to implement a fast (probabilistic) prime testing like the Miller-Rabin's algorithm (see **Exercise 5.3.2.4***).

```cpp
#define abs_val(a) (((a)>0)?(a):-(a))
typedef long long ll;

ll mulmod(ll a, ll b, ll c) { // returns (a * b) % c, and minimize overflow
  ll x = 0, y = a % c;
  while (b > 0) {
    if (b % 2 == 1) x = (x + y) % c;
    y = (y * 2) % c;
    b /= 2;
  }
  return x % c;
}


ll gcd(ll a,ll b) { return !b ? a : gcd(b, a % b); }        // standard gcd

ll pollard_rho(ll n) {
  int i = 0, k = 2;
  ll x = 3, y = 3;                    // random seed = 3, other values possible
  while (1) {
    i++;
    x = (mulmod(x, x, n) + n - 1) % n;                  // generating function
    ll d = gcd(abs_val(y - x), n);                        // the key insight
    if (d != 1 && d != n) return d;        // found one non-trivial factor
    if (i == k) y = x, k *= 2;
} }

int main() {
  ll n = 2063512844981574047LL;     // we assume that n is not a large prime
  ll ans = pollard_rho(n);             // break n into two non trivial factors
  if (ans > n / ans) ans = n / ans;          // make ans the smaller factor
  printf("%lld %lld\n", ans, n / ans);  // should be: 1112041493 1855607779
} // return 0;
```

We can also implement Pollard's rho algorithm in Java and use the `isProbablePrime` function in Java BigInteger class. This way, we can accept n larger than $2^{64} - 1$, e.g. 17798655664295576020099, which is $\approx 2^{74}$, and factor it into $143054969437 \times 124418296927$. However, the runtime of Pollard's rho algorithm increases with larger n. The fact that integer factoring is a very difficult task is still the key concept of modern cryptography.

It is a good idea to test the complete implementation of Pollard's rho algorithm (that is, including the fast probabilistic prime testing algorithm and any other small details) to solve the following two programming exercise problems.

Source code: `Pollardsrho.cpp/java`

Programming exercises related to Pollard's rho algorithm:

1. **_UVa 11476 - Factoring Large(t) ... *_** (see the discussion above)
2. POJ 1811 - Prime Test, see `http://poj.org/problem?id=1811`

9.27 Postfix Calculator and Conversion

Algebraic Expressions

There are three types of algebraic expressions: Infix (the natural way for human to write algebraic expressions), Prefix[24] (Polish notation), and Postfix (Reverse Polish notation). In Infix/Prefix/Postfix expressions, an operator is located (in the middle of)/before/after two operands, respectively. In Table 9.2, we show three Infix expressions, their corresponding Prefix/Postfix expressions, and their values.

Infix	Prefix	Postfix	Value
2 + 6 * 3	+ 2 * 6 3	2 6 3 * +	20
(2 + 6) * 3	* + 2 6 3	2 6 + 3 *	24
4 * (1 + 2 * (9 / 3) - 5)	* 4 - + 1 * 2 / 9 3 5	4 1 2 9 3 / * + 5 - *	8

Table 9.2: Examples of Infix, Prefix, and Postfix expressions

Postfix Calculator

Postfix expressions are more computationally efficient than Infix expressions. First, we do not need (complex) parentheses as the precedence rules are already embedded in the Postfix expression. Second, we can also compute partial results as soon as an operator is specified. These two features are not found in Infix expressions.

Postfix expression can be computed in $O(n)$ using Postfix calculator algorithm. Initially, we start with an empty stack. We read the expression from left to right, one token at a time. If we encounter an operand, we will push it to the stack. If we encounter an operator, we will pop the top two items of the stack, do the required operation, and then put the result back to the stack. Finally, when all tokens have been read, we return the top (the only item) of the stack as the final answer.

As each of the n tokens is only processed once and all stack operations are $O(1)$, this Postfix Calculator algorithm runs in $O(n)$.

An example of a Postfix calculation is shown in Table 9.3.

Postfix	Stack (bottom to top)	Remarks
<u>4 1 2 9 3</u> / * + 5 - *	4 1 2 9 3	The first five tokens are operands
4 1 2 9 3 <u>/</u> * + 5 - *	4 1 2 3	Take 3 and 9, compute 9 / 3, push 3
4 1 2 9 3 / <u>*</u> + 5 - *	4 1 6	Take 3 and 2, compute 2 * 3, push 6
4 1 2 9 3 / * <u>+</u> 5 - *	4 7	Take 6 and 1, compute 1 + 6, push 7
4 1 2 9 3 / * + <u>5</u> - *	4 7 5	An operand
4 1 2 9 3 / * + 5 <u>-</u> *	4 7 5	Take 5 and 7, compute 7 - 5, push 2
4 1 2 9 3 / * + 5 - <u>*</u>	4 2	Take 2 and 4, compute 4 * 2, push 8
4 1 2 9 3 / * + 5 - *	8	Return 8 as the answer

Table 9.3: Example of a Postfix Calculation

Exercise 9.27.1*: What if we are given Prefix expressions instead? How to evaluate a Prefix expression in $O(n)$?

---

[24]One programming language that uses this expression is Scheme.

Infix to Postfix Conversion

Knowing that Postfix expressions are more computationally efficient than Infix expressions, many compilers will convert Infix expressions in the source code (most programming languages use Infix expressions) into Postfix expressions. To use the efficient Postfix Calculator as shown earlier, we need to be able to convert Infix expressions into Postfix expressions efficiently. One of the possible algorithm is the 'Shunting yard' algorithm invented by Edsger Dijkstra (the inventor of Dijkstra's algorithm—see Section 4.4.3).

Shunting yard algorithm has similar flavor with Bracket Matching (see Section 9.4) and Postfix Calculator above. The algorithm also uses a stack, which is initially empty. We read the expression from left to right, one token at a time. If we encounter an operand, we will immediately output it. If we encounter an open bracket, we will push it to the stack. If we encounter a close bracket, we will output the topmost items of the stack until we encounter an open bracket (but we do not output the open bracket). If we encounter an operator, we will keep outputting and then popping the topmost item of the stack if it has greater than or equal precedence with this operator, or until we encounter an open bracket, then push this operator to the stack. At the end, we will keep outputting and then popping the topmost item of the stack until the stack is empty.

As each of the n tokens is only processed once and all stack operations are $O(1)$, this Shunting yard algorithm runs in $O(n)$.

An example of a Shunting yard algorithm execution is shown in Table 9.4.

Infix	Stack	Postfix	Remarks
<u>4</u> * (1 + 2 * (9 / 3) - 5)		4	Immediately output
4 <u>*</u> (1 + 2 * (9 / 3) - 5)	*	4	Put to stack
4 * <u>(</u> 1 + 2 * (9 / 3) - 5)	* (	4	Put to stack
4 * (<u>1</u> + 2 * (9 / 3) - 5)	* (	4 1	Immediately output
4 * (1 <u>+</u> 2 * (9 / 3) - 5)	* (+	4 1	Put to stack
4 * (1 + <u>2</u> * (9 / 3) - 5)	* (+	4 1 2	Immediately output
4 * (1 + 2 <u>*</u> (9 / 3) - 5)	* (+ *	4 1 2	Put to stack
4 * (1 + 2 * <u>(</u> 9 / 3) - 5)	* (+ * (	4 1 2	Put to stack
4 * (1 + 2 * (<u>9</u> / 3) - 5)	* (+ * (	4 1 2 9	Immediately output
4 * (1 + 2 * (9 <u>/</u> 3) - 5)	* (+ * (/	4 1 2 9	Put to stack
4 * (1 + 2 * (9 / <u>3</u>) - 5)	* (+ * (/	4 1 2 9 3	Immediately output
4 * (1 + 2 * (9 / 3 <u>)</u> - 5)	* (+ *	4 1 2 9 3 /	Only output '/'
4 * (1 + 2 * (9 / 3) <u>-</u> 5)	* (-	4 1 2 9 3 / * +	Output '*' then '+'
4 * (1 + 2 * (9 / 3) - <u>5</u>)	* (-	4 1 2 9 3 / * + 5	Immediately output
4 * (1 + 2 * (9 / 3) - 5 <u>)</u>	*	4 1 2 9 3 / * + 5 -	Only output '-'
4 * (1 + 2 * (9 / 3) - 5 <u>)</u>		4 1 2 9 3 / * + 5 - *	Empty the stack

Table 9.4: Example of an Execution of Shunting yard Algorithm

Programming exercises related to Postfix expression:

1. **UVa 00727 - Equation *** (the classic Infix to Postfix conversion problem)

9.28   Roman Numerals

Problem Description

Roman Numerals is a number system used in ancient Rome. It is actually a Decimal number system but it uses a certain letters of the alphabet instead of digits [0..9] (described below), it is not positional, and it does not have a symbol for zero.

Roman Numerals have these 7 basic letters and its corresponding Decimal values: I=1, V=5, X=10, L=50, C=100, D=500, and M=1000. Roman Numerals also have the following letter pairs: IV=4, IX=9, XL=40, XC=90, CD=400, CM=900.

Programming problems involving Roman Numerals usually deal with the conversion from Arabic numerals (the Decimal number system that we normally use everyday) to Roman Numerals and vice versa. Such problems only appear very rarely in programming contests and such conversion can be derived on the spot by reading the problem statement.

Solution(s)

In this section, we provide one conversion library that we have used to solve several programming problems involving Roman Numerals. Although you can derive this conversion code easily, at least you do not have to debug[25] if you already have this library.

```
void AtoR(int A) {
  map<int, string> cvt;
  cvt[1000] = "M"; cvt[900] = "CM"; cvt[500] = "D"; cvt[400] = "CD";
  cvt[100]  = "C"; cvt[90]  = "XC"; cvt[50]  = "L"; cvt[40]   = "XL";
  cvt[10]   = "X"; cvt[9]   = "IX"; cvt[5]   = "V"; cvt[4]    = "IV";
  cvt[1]    = "I";
  // process from larger values to smaller values
  for (map<int, string>::reverse_iterator i = cvt.rbegin();
       i != cvt.rend(); i++)
    while (A >= i->first) {
      printf("%s", ((string)i->second).c_str());
      A -= i->first; }
  printf("\n");
}


void RtoA(char R[]) {
  map<char, int> RtoA;
  RtoA['I'] = 1;   RtoA['V'] = 5;   RtoA['X'] = 10;   RtoA['L'] = 50;
  RtoA['C'] = 100; RtoA['D'] = 500; RtoA['M'] = 1000;

  int value = 0;
  for (int i = 0; R[i]; i++)
    if (R[i+1] && RtoA[R[i]] < RtoA[R[i+1]]) {     // check next char first
      value += RtoA[R[i + 1]] - RtoA[R[i]];                  // by definition
      i++; }                                                 // skip this char
    else value += RtoA[R[i]];
  printf("%d\n", value);
}
```

---

[25]If the problem uses different standard of Roman Numerals, you may need to slightly edit our code.

Source code: `UVa11616.cpp/java`

Programming exercises related to Roman Numerals:

1. **UVa 00344 - Roman Digititis *** (count how many Roman characters are used to make all numbers from 1 to N)

2. UVa 00759 - The Return of the ... (Roman number + validity check)

3. **UVa 11616 - Roman Numerals *** (Roman numeral conversion problem)

4. ***UVa 12397 - Roman Numerals **** (conversion, each Roman digit has value)

9.29 Selection Problem

Problem Description

Selection problem is the problem of finding the k-th smallest[26] element of an array of n elements. Another name for selection problem is order statistics. Thus the minimum (smallest) element is the 1-st order statistic, the maximum (largest) element is the n-th order statistic, and the median element is the $\frac{n}{2}$ order statistic (there are 2 medians if n is even).

This selection problem is used as a motivating example in the opening of Chapter 3. In this section, we discuss this problem, its variants, and its various solutions in more details.

Solution(s)

Special Cases: $k = 1$ and $k = n$

Searching the minimum ($k = 1$) or maximum ($k = n$) element of an arbitrary array can be done in $\Omega(n - 1)$ comparisons: We set the first element to be the temporary answer, and then we compare this temporary answer with the other $n - 1$ elements one by one and keep the smaller (or larger, depending on the requirement) one. Finally, we report the answer. $\Omega(n - 1)$ comparisons is the lower bound, i.e. We cannot do better than this. While this problem is easy for $k = 1$ or $k = n$, finding the other order statistics—the general form of selection problem—is more difficult.

$O(n^2)$ algorithm, static data

A naïve algorithm to find the k-th smallest element is to this: Find the smallest element, 'discard' it (e.g. by setting it to a 'dummy large value'), and repeat this process k times. When k is near 1 (or when k is near n), this $O(kn)$ algorithm can still be treated as running in $O(n)$, i.e. we treat k as a 'small constant'. However, the worst case scenario is when we have to find the median ($k = \frac{n}{2}$) element where this algorithm runs in $O(\frac{n}{2} \times n) = O(n^2)$.

$O(n \log n)$ algorithm, static data

A better algorithm is to sort (that is, pre-process) the array first in $O(n \log n)$. Once the array is sorted, we can find the k-th smallest element in $O(1)$ by simply returning the content of index k-1 (0-based indexing) of the sorted array. The main part of this algorithm is the sorting phase. Assuming that we use a good $O(n \log n)$ sorting algorithm, this algorithm runs in $O(n \log n)$ overall.

Expected $O(n)$ algorithm, static data

An even better algorithm for the selection problem is to apply Divide and Conquer paradigm. The key idea of this algorithm is to use the $O(n)$ Partition algorithm (the randomized version) from Quick Sort as its sub-routine.

A randomized partition algorithm: `RandomizedPartition(A, l, r)` is an algorithm to partition a given range `[l..r]` of the array A around a (random) pivot. Pivot $A[p]$ is one of the element of A where $p \in$ `[l..r]`. After partition, all elements $\leq A[p]$ are placed before the pivot and all elements $> A[p]$ are placed after the pivot. The final index of the pivot q is returned. This randomized partition algorithm can be done in $O(n)$.

---

[26]Note that finding the k-th largest element is equivalent to finding the $(n$-k+1)-th smallest element.

After performing q = RandomizedPartition(A, 0, n - 1), all elements $\leq A[q]$ will be placed before the pivot and therefore $A[q]$ is now in it's correct order statistic, which is $q+1$. Then, there are only 3 possibilities:

1. $q + 1 = k$, $A[q]$ is the desired answer. We return this value and stop.

2. $q + 1 > k$, the desired answer is inside the left partition, e.g. in $A[0..q\text{-}1]$.

3. $q + 1 < k$, the desired answer is inside the right partition, e.g. in $A[q\text{+}1..n\text{-}1]$.

This process can be repeated recursively on smaller range of search space until we find the required answer. A snippet of C++ code that implements this algorithm is shown below.

```
int RandomizedSelect(int A[], int l, int r, int k) {
  if (l == r) return A[l];
  int q = RandomizedPartition(A, l, r);
      if (q + 1 == k) return A[q];
  else if (q + 1  > k) return RandomizedSelect(A, l, q - 1, k);
  else                 return RandomizedSelect(A, q + 1, r, k);
}
```

This RandomizedSelect algorithm runs in expected $O(n)$ time and very unlikely to run in its worst case $O(n^2)$ as it uses randomized pivot at each step. The full analysis involves probability and expected values. Interested readers are encouraged to read other references for the full analysis e.g. [7].

A simplified (but not rigorous) analysis is to assume RandomizedSelect divides the array into two at each step and n is a power of two. Therefore it runs RandomizedPartition in $O(n)$ for the first round, in $O(\frac{n}{2})$ in the second round, in $O(\frac{n}{4})$ in the third round and finally $O(1)$ in the $1 + \log_2 n$ round. The cost of RandomizedSelect is mainly determined by the cost of RandomizedPartition as all other steps of RandomizedSelect is $O(1)$. Therefore the overall cost is $O(n + \frac{n}{2} + \frac{n}{4} + ... + \frac{n}{n}) = O(n \times (\frac{1}{1} + \frac{1}{2} + \frac{1}{4} + ... + \frac{1}{n}) \leq O(2n) = O(n)$.

Library solution for the expected $O(n)$ algorithm, static data

C++ STL has function nth_element in <algorithm>. This nth_element implements the expected $O(n)$ algorithm as shown above. However as of 24 May 2013, we are not aware of Java equivalent for this function.

$O(n \log n)$ pre-processing, $O(\log n)$ algorithm, dynamic data

All solutions presented earlier assume that the given array is static—unchanged for each query of the k-th smallest element. However, if the content of the array is frequently modified, i.e. a new element is added, an existing element is removed, or the value of an existing element is changed, the solutions outlined above become inefficient.

When the underlying data is dynamic, we need to use a balanced Binary Search Tree (see Section 2.3). First, we insert all n elements into a balanced BST in $O(n \log n)$ time. We also augment (add information) about the size of each sub-tree rooted at each vertex. This way, we can find the k-th smallest element in $O(\log n)$ time by comparing k with q—the size of the left sub-tree of the root:

1. If $q + 1 = k$, then the root is the desired answer. We return this value and stop.

2. If $q + 1 > k$, the desired answer is inside the left sub-tree of the root.

3. If $q + 1 < k$, the desired answer is inside the right sub-tree of the root and we are now searching for the $(k - q - 1)$-th smallest element in this right sub-tree. This adjustment of k is needed to ensure correctness.

This process—which is similar with the expected $O(n)$ algorithm for static selection problem—can be repeated recursively until we find the required answer. As checking the size of a sub-tree can be done in $O(1)$ if we have properly augment the BST, this overall algorithm runs at worst in $O(\log n)$ time, from root to the deepest leaf of a balanced BST.

However, as we need to augment a balanced BST, this algorithm cannot use built-in C++ STL <map>/<set> (or Java `TreeMap`/`TreeSet`) as these library code cannot be augmented. Therefore, we need to write our own balanced BST routine (e.g. AVL tree—see Figure 9.12—or Red Black Tree, etc—all of them take some time to code) and therefore such selection problem on *dynamic data* can be quite painful to solve.
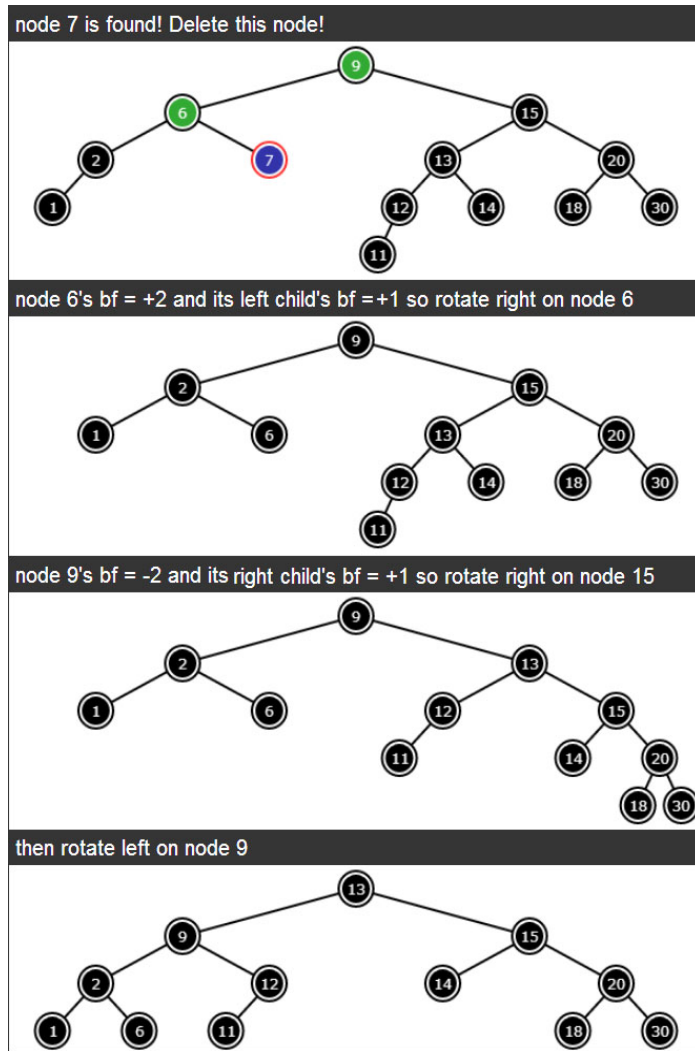
Figure 9.12: Example of an AVL Tree Deletion (Delete 7)

Visualization: `www.comp.nus.edu.sg/~stevenha/visualization/bst.html`

9.30 Shortest Path Faster Algorithm

Shortest Path Faster Algorithm (SPFA) is an algorithm that utilizes a queue to eliminate redundant operations in Bellman Ford's algorithm. This algorithm was published in Chinese by Duan Fanding in 1994. As of 2013, this algorithm is popular among Chinese programmers but it is not yet well known in other parts of the world.

SPFA requires the following data structures:

1. A graph stored in an Adjacency List: `AdjList` (see Section 2.4.1).

2. `vi dist` to record the distance from source to every vertex.
   (`vi` is our shortcut for `vector<int>`).

3. A `queue<int>` to stores the vertex to be processed.

4. `vi in_queue` to denote if a vertex is in the queue or not.

The first three data structures are the same as Dijkstra's or Bellman Ford's algorithms listed in Section 4.4. The fourth data structure is unique to SPFA. We can write SPFA as follows:

```
// inside int main()
  // initially, only S has dist = 0 and in the queue
  vi dist(n, INF); dist[S] = 0;
  queue<int> q; q.push(S);
  vi in_queue(n, 0); in_queue[S] = 1;

  while (!q.empty()) {
    int u = q.front(); q.pop(); in_queue[u] = 0;
    for (j = 0; j < (int)AdjList[u].size(); j++) {    // all neighbors of u
      int v = AdjList[u][j].first, weight_u_v = AdjList[u][j].second;
      if (dist[u] + weight_u_v < dist[v]) {                 // if can relax
        dist[v] = dist[u] + weight_u_v;                           // relax
        if (!in_queue[v]) {                           // add to the queue
          q.push(v);              // only if it is not already in the queue
          in_queue[v] = 1;
  } } } }
```

Source code: `UVa10986.cpp/java`

This algorithm runs in $O(kE)$ where k is a number depending on the graph. The maximum k can be V (which is the same as the time complexity of Bellman Ford's). However, we have tested that for most SSSP problems in UVa online judge that are listed in this book, SPFA (which uses a queue) is as fast as Dijkstra's (which uses a priority queue).

SPFA can deal with negative weight edge. If the graph has no negative cycle, SPFA runs well on it. If the graph has negative cycle(s), SPFA can also detect it as there must be some vertex (those on the negative cycle) that enters the queue for over $V - 1$ times. We can modify the given code above to record the time each vertex enters the queue. If we find that any vertex enters the queue more than $V - 1$ times, we can conclude that the graph has negative cycle(s).

9.31 Sliding Window

Problem Description

There are several variants of Sliding Window problems. But all of them have similar basic idea: 'Slide' a sub-array (that we call a 'window', which can have static or dynamic length) in linear fashion from left to right over the original array of n elements in order to compute something. Some of the variants are:

1. Find the smallest sub-array size (smallest window length) so that the sum of the sub-array is greater than or equal to a certain constant S in $O(n)$? Examples:
   For array $A_1 = \{5, 1, 3, [5, 10], 7, 4, 9, 2, 8\}$ and $S = 15$, the answer is 2 as highlighted.
   For array $A_2 = \{1, 2, [3, 4, 5]\}$ and $S = 11$, the answer is 3 as highlighted.

2. Find the smallest sub-array size (smallest window length) so that the elements inside the sub-array contains all integers in range [1..K]. Examples:
   For array $A = \{1, [2, 3, 7, 1, 12, 9, 11, 9, 6, 3, 7, 5, 4], 5, 3, 1, 10, 3, 3\}$ and $K = 4$, the answer is 13 as highlighted.
   For the same array $A = \{[1, 2, 3], 7, 1, 12, 9, 11, 9, 6, 3, 7, 5, 4, 5, 3, 1, 10, 3, 3\}$ and $K = 3$, the answer is 3 as highlighted.

3. Find the maximum sum of a certain sub-array with (static) size K. Examples:
   For array $A_1 = \{10, [50, 30, 20], 5, 1\}$ and $K = 3$, the answer is 100 by summing the highlighted sub-array.
   For array $A_2 = \{49, 70, 48, [61, 60], 60\}$ and $K = 2$, the answer is 121 by summing the highlighted sub-array.

4. Find the minimum of *each* possible sub-arrays with (static) size K. Example:
   For array $A = \{0, 5, 5, 3, 10, 0, 4\}$, $n = 7$, and $K = 3$, there are $n - K + 1 = 7 - 3 + 1 = 5$ possible sub-arrays with size $K = 3$, i.e. {0, 5, 5}, {5, 5, 3}, {5, 3, 10}, {3, 10, 0}, and {10, 0, 4}. The minimum of each sub-array is 0, 3, 3, 0, 0, respectively.

Solution(s)

We ignore the discussion of naïve solutions for these Sliding Window variants and go straight to the $O(n)$ solutions to save space. The four solutions below run in $O(n)$ as what we do is to 'slide' a window over the original array of n elements—some with clever tricks.

For variant number 1, we maintain a window that keeps growing (append the current element to the back—the right side—of the window) and add the value of the current element to a running sum or keeps shrinking (remove the front—the left side—of the window) as long as the running sum is $\geq S$. We keep the smallest window length throughout the process and report the answer.

For variant number 2, we maintain a window that keeps growing if range [1..K] is not yet covered by the elements of the current window or keeps shrinking otherwise. We keep the smallest window length throughout the process and report the answer. The check whether range [1..K] is covered or not can be simplified using a kind of frequency counting. When all integers ∈ [1..K] has non zero frequency, we said that range [1..K] is covered. Growing the window increases a frequency of a certain integer that may cause range [1..K] to be fully covered (it has no 'hole') whereas shrinking the window decreases a frequency of the removed integer and if the frequency of that integer drops to 0, the previously covered range [1..K] is now no longer covered (it has a 'hole').

For variant number 3, we insert the first K integers into the window, compute its sum, and declare the sum as the current maximum. Then we slide the window to the right by adding one element to the right side of the window and removing one element from the left side of the window—thereby maintaining window length to K. We add the sum by the value of the added element minus the value of the removed element and compare with the current maximum sum to see if this sum is the new maximum sum. We repeat this window-sliding process $n - K$ times and report the maximum sum found.

Variant number 4 is quite challenging especially if n is large. To get $O(n)$ solution, we need to use a deque (double-ended queue) data structure to model the window. This is because deque supports efficient—$O(1)$—insertion and deletion from front and back of the queue (see discussion of deque in Section 2.2). This time, we maintain that the window (that is, the deque) is sorted in ascending order, that is, the front most element of the deque has the minimum value. However, this changes the ordering of elements in the array. To keep track of whether an element is currently still inside the current window or not, we need to remember the index of each element too. The detailed actions are best explained with the C++ code below. This sorted window can shrink from both sides (back and front) and can grow from back, thus necessitating the usage of deque[27] data structure.

```
void SlidingWindow(int A[], int n, int K) {
  // ii---or pair<int, int>---represents the pair (A[i], i)
  deque<ii> window; // we maintain 'window' to be sorted in ascending order
  for (int i = 0; i < n; i++) {                          // this is O(n)
    while (!window.empty() && window.back().first >= A[i])
      window.pop_back();                  // this to keep 'window' always sorted

    window.push_back(ii(A[i], i));

    // use the second field to see if this is part of the current window
    while (window.front().second <= i - K)              // lazy deletion
      window.pop_front();

    if (i + 1 >= K)            // from the first window of length K onwards
      printf("%d\n", window.front().first);  // the answer for this window
} }
```

Programming exercises:

1. **UVa 01121 - Subsequence *** (sliding window variant no 1)

2. **UVa 11536 - Smallest Sub-Array *** (sliding window variant no 2)

3. IOI 2011 - Hottest (practice task; sliding window variant no 3)

4. IOI 2011 - Ricehub (sliding window++)

5. IOI 2012 - Tourist Plan (practice task; another sliding window variant; the best answer starting from city 0 and ending at city $i \in [0..N\text{-}1]$ is the sum of happiness of the top $K\text{-}i$ cities $\in [0..i]$; use priority_queue; output the highest sum)

---

[27]Note that we do not actually need to use deque data structure for variant 1-3 above.

9.32 Sorting in Linear Time

Problem Description

Given an (unsorted) array of n elements, can we sort them in $O(n)$ time?

Theoretical Limit

In general case, the lower bound of generic—comparison-based—sorting algorithm is $\Omega(n \log n)$ (see the proof using decision tree model in other references, e.g. [7]). However, if there is a special property about the n elements, we can have a faster, linear, $O(n)$ sorting algorithm by *not* doing comparison between elements. We will see two examples below.

Solution(s)

Counting Sort

If the array `A` contains n integers with *small* range `[L..R]` (e.g. 'human age' of `[1..99]` years in UVa 11462 - Age Sort), we can use the Counting Sort algorithm. For the explanation below, assume that array `A` is {2, 5, 2, 2, 3, 3}. The idea of Counting Sort is as follows:

1. Prepare a 'frequency array' `f` with size `k = R-L+1` and initialize `f` with zeroes.

   On the example array above, we have `L = 2`, `R = 5`, and `k = 4`.

2. We do one pass through array `A` and update the frequency of each integer that we see, i.e. for each `i` ∈ `[0..n-1]`, we do `f[A[i]-L]++`.

   On the example array above, we have `f[0] = 3`, `f[1] = 2`, `f[2] = 0`, `f[3] = 1`.

3. Once we know the frequency of each integers in that small range, we compute the prefix sums of each `i`, i.e. `f[i] = [f-1] + f[i]` ∀`i` ∈ `[1..k-1]`. Now, `f[i]` contains the number of elements less than or equal to `i`.

   On the example array above, we have `f[0] = 3`, `f[1] = 5`, `f[2] = 5`, `f[3] = 6`.

4. Next, go backwards from `i = n-1` down to `i = 0`.
   We place `A[i]` at index `f[A[i]-L]-1` as it is the correct location for `A[i]`.
   We decrement `f[A[i]-L]` by one so that the next copy of `A[i]`—if any—will be placed right before the current `A[i]`.

   On the example array above, we first put `A[5] = 3` in index `f[A[5]-2]-1 = f[1]-1 = 5-1 = 4` and decrement `f[1]` to 4.
   Next, we put `A[4] = 3`—the same value as `A[5] = 3`—now in index `f[A[4]-2]-1 = f[1]-1 = 4-1 = 3` and decrement `f[1]` to 3.
   Then, we put `A[3] = 2` in index `f[A[3]-2]-1 = 2` and decrement `f[0]` to 2.
   We repeat the next three steps until we obtain a sorted array: {2, 2, 2, 3, 3, 5}.

The time complexity of Counting Sort is $O(n + k)$. When $k = O(n)$, this algorithm theoretically runs in linear time by *not* doing comparison of the integers. However, in programming contest environment, usually k cannot be too large in order to avoid Memory Limit Exceeded. For example, Counting Sort will have problem sorting this array `A` with $n = 3$ that contains {1, 1000000000, 2} as it has large k.

Radix Sort

If the array `A` contains n non-negative integers with relatively wide range `[L..R]` but it has relatively small number of digits, we can use the Radix Sort algorithm.

The idea of Radix Sort is simple. First, we make all integers have d digits—where d is the largest number of digits in the largest integer in `A`—by appending zeroes if necessary. Then, Radix Sort will sort these numbers digit by digit, starting with the *least* significant digit to the *most* significant digit. It uses another *stable sort* algorithm as a sub-routine to sort the digits, such as the $O(n + k)$ Counting Sort shown above. For example:

```
Input  | Append | Sort by the  | Sort by the | Sort by the  | Sort by the
d = 4  | Zeroes | fourth digit | third digit | second digit | first digit
 323   | 0323   | 032(2)       | 00(1)3      | 0(0)13       | (0)013
1257   | 1257   | 032(3)       | 03(2)2      | 1(2)57       | (0)322
  13   | 0013   | 001(3)       | 03(2)3      | 0(3)22       | (0)323
 322   | 0322   | 125(7)       | 12(5)7      | 0(3)23       | (1)257
```

For an array of n d-digits integers, we will do an $O(d)$ passes of Counting Sorts which have time complexity of $O(n + k)$ each. Therefore, the time complexity of Radix Sort is $O(d \times (n + k))$. If we use Radix Sort for sorting n 32-bit signed integers ($\approx d = 10$ digits) and k = 10. This Radix Sort algorithm runs in $O(10 \times (n + 10))$. It can still be considered as running in linear time but it has high constant factor.

Considering the hassle of writing the complex Radix Sort routine compared to calling the standard $O(n \log n)$ C++ STL `sort` (or Java `Collections.sort`), this Radix Sort algorithm is rarely used in programming contests. In this book, we only use this combination of Radix Sort and Counting Sort in our Suffix Array implementation (see Section 6.6.4).

Exercise 9.32.1*: What should we do if we want to use Radix Sort but the array `A` contains (at least one) negative number(s)?

Programming exercises related to Sorting in Linear Time:

1. **UVa 11462 - Age Sort *** (standard Counting Sort problem)

9.33 Sparse Table Data Structure

In Section 2.4.3, we have seen that Segment Tree data structure can be used to solve the Range Minimum Query (RMQ) problem—the problem of finding the index that has the minimum element within a range [i..j] of the underlying array A. It takes $O(n)$ pre-processing time to build the Segment Tree, and once the Segment Tree is ready, each RMQ is just $O(\log n)$. With Segment Tree, we can deal with the *dynamic version* of this RMQ problem, i.e. when the underlying array is updated, we usually only need $O(\log n)$ to update the corresponding Segment Tree structure.

However, some problems involving RMQ never change the underlying array A after the first query. This is called the *static* RMQ problem. Although Segment Tree obviously can be used to deal with the static RMQ problem, this static version has an alternative DP solution with $O(n \log n)$ pre-processing time and $O(1)$ per RMQ. One such example is the Lowest Common Ancestor (LCA) problem in Section 9.18.

The key idea of the DP solution is to split A into sub arrays of length 2^j for each non-negative integer j such that $2^j \le n$. We will keep an array SpT of size $n \times \log n$ where SpT[i][j] stores the index of the minimum value in the sub array starting at index i and having length 2^j. This array SpT will be sparse as not all of its cells have values (hence the name 'Sparse Table'). We use an abbreviation SpT to differentiate this data structure from Segment Tree (ST).

To build up the SpT array, we use a technique similar to the one used in many Divide and Conquer algorithms such as merge sort. We know that in an array of length 1, the single element is the smallest one. This is our base case. To find out the index of the smallest element in an array of size 2^j, we can compare the values at the indices of the smallest elements in the two distinct sub arrays of size 2^{j-1} and take the index of the smallest element of the two. It takes $O(n \log n)$ time to build up the SpT array like this. Please scrutinize the constructor of class RMQ shown in the source code below that implements this SpT array construction.

It is simple to understand how we would process a query if the length of the range were a power of 2. Since this is exactly the information SpT stores, we would just return the corresponding entry in the array. However, in order to compute the result of a query with arbitrary start and end indices, we have to fetch the entry for two smaller sub arrays within this range and take the minimum of the two. Note that these two sub arrays might have to overlap, the point is that we want cover the entire range with two sub arrays and nothing outside of it. This is always possible even if the length of the sub arrays have to be a power of 2. First, we find the length of the query range, which is j-i+1. Then, we apply $\log_2$ on it and round down the result, i.e. k = $\lfloor \log_2(\text{j-i+1}) \rfloor$. This way, $2^k \le$ (j-i+1). This simple Figure 9.13 below shows what the two sub arrays might look like. As there is a potentially overlapping sub-problems, this part of the solution is classified as Dynamic Programming.
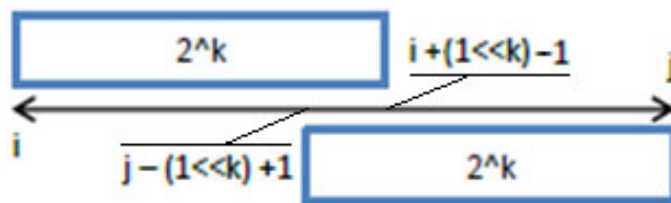
Figure 9.13: Explanation of RMQ(i, j)

An example implementation of Sparse Table to solve the static RMQ problem is shown below. You can compare this version with the Segment Tree version shown in Section 2.4.3.

```
#define MAX_N 1000                                    // adjust this value as needed
#define LOG_TWO_N 10                 // 2^10 > 1000, adjust this value as needed

class RMQ {                                            // Range Minimum Query
private:
  int _A[MAX_N], SpT[MAX_N][LOG_TWO_N];
public:
  RMQ(int n, int A[]) {     // constructor as well as pre-processing routine
    for (int i = 0; i < n; i++) {
      _A[i] = A[i];
      SpT[i][0] = i; // RMQ of sub array starting at index i + length 2^0=1
    }
    // the two nested loops below have overall time complexity = O(n log n)
    for (int j = 1; (1<<j) <= n; j++) // for each j s.t. 2^j <= n, O(log n)
      for (int i = 0; i + (1<<j) - 1 < n; i++)     // for each valid i, O(n)
        if (_A[SpT[i][j-1]] < _A[SpT[i+(1<<(j-1))][j-1]])          // RMQ
          SpT[i][j] = SpT[i][j-1];     // start at index i of length 2^(j-1)
        else                      // start at index i+2^(j-1) of length 2^(j-1)
          SpT[i][j] = SpT[i+(1<<(j-1))][j-1];
  }

  int query(int i, int j) {                              // this query is O(1)
    int k = (int)floor(log((double)j-i+1) / log(2.0));     // 2^k <= (j-i+1)
    if (_A[SpT[i][k]] <= _A[SpT[j-(1<<k)+1][k]]) return SpT[i][k];
    else                                       return SpT[j-(1<<k)+1][k];
} };
```

Source code: `SparseTable.cpp/java`

For the same test case with $n = 7$ and $A = \{18, 17, 13, 19, 15, 11, 20\}$ as in Section 2.4.3, the content of the sparse table SpT is as follows:

index	0	1	2
0	0	1	2
1	1	2	2
2	2	2	5
3	3	4	5
4	4	5	empty
5	5	5	empty
6	6	empty	empty

In the first column, we have $j = 0$ that denotes the RMQ of sub array starting at index i with length $2^0 = 1$, we have SpT[i][j] = i.

In the second column, we have $j = 1$ that denotes the RMQ of sub array starting at index i with length $2^1 = 2$. Notice that the last row is empty.

In the third column, we have $j = 2$ that denotes the RMQ of sub array starting at index i with length $2^2 = 4$. Notice that the last three rows is empty.

9.34 Tower of Hanoi

Problem Description

The classic description of the problem is as follows: There are three pegs: A, B, and C, as well as n discs, will all discs having different sizes. Starting with all the discs stacked in ascending order on one peg (peg A), your task is to move all n discs to another peg (peg C). No disc may be placed on top of a disc smaller than itself, and only one disc can be moved at a time, from the top of one peg to another.

Solution(s)

There exists a simple recursive backtracking solution for the classic Tower of Hanoi problem. The problem of moving n discs from peg A to peg C with additional peg B as intermediate peg can be broken up into the following sub-problems:

1. Move $n - 1$ discs from peg A to peg B using peg C as the intermediate peg.
   After this recursive step is done, we are left with disc n by itself in peg A.

2. Move disc n from peg A to peg C.

3. Move $n - 1$ discs from peg B to peg C using peg A as the intermediate peg.
   These $n - 1$ discs will be on top of disc n which is now at the bottom of peg C.

Note that step 1 and step 3 above are recursive steps. The base case is when $n = 1$ where we simply move a single disc from the current source peg to its destination peg, bypassing the intermediate peg. A sample C++ implementation code is shown below:

```cpp
#include <cstdio>
using namespace std;

void solve(int count, char source, char destination, char intermediate) {
  if (count == 1)
    printf("Move top disc from pole %c to pole %c\n", source, destination);
  else {
    solve(count-1, source, intermediate, destination);
    solve(1, source, destination, intermediate);
    solve(count-1, intermediate, destination, source);
  }
}

int main() {
  solve(3, 'A', 'C', 'B');      // try larger value for the first parameter
} // return 0;
```

The minimum number of moves required to solve a classic Tower of Hanoi puzzle of n discs using this recursive backtracking solution is $2^n - 1$ moves.

Programming exercises related to Tower of Hanoi:

1. **UVa 10017 - The Never Ending ... *** (classical problem)

9.35   Chapter Notes

As of 24 May 2013, Chapter 9 contains 34 rare topics. 10 of them are rare algorithms (highlighted in **bold**). The other 24 are rare problems.

2-SAT Problem	Art Gallery Problem
Bitonic Traveling Salesman Problem	Bracket Matching
Chinese Postman Problem	Closest Pair Problem
Dinic's Algorithm	**Formulas or Theorems**
Gaussian Elimination Algorithm	Graph Matching
Great-Circle Distance	**Hopcroft Karp's Algorithm**
Independent and Edge-Disjoint Paths	Inversion Index
Josephus Problem	Knight Moves
Kosaraju's Algorithm	Lowest Common Ancestor
Magic Square Construction (Odd Size)	Matrix Chain Multiplication
Matrix Power	Max Weighted Independent Set
Min Cost (Max) Flow	Min Path Cover on DAG
Pancake Sorting	**Pollard's rho Integer Factoring Algorithm**
Postfix Calculator and Conversion	Roman Numerals
Selection Problem	Shortest Path Faster Algorithm
Sliding Window	Sorting in Linear Time
Sparse Table Data Structure	Tower of Hanoi

However, after writing so much in the third edition of this book, we become more aware that there are many other Computer Science topics that we have not covered yet.

We close this chapter—and the third edition of this book—by listing down quite a good number of topic keywords that are eventually not included in the third edition of this book due to our-own self-imposed 'writing time limit' of 24 May 2013.

There are many other exotic data structures that are rarely used in programming contests: Fibonacci heap, various hashing techniques (hash tables), heavy-light decomposition of a rooted tree, interval tree, k-d tree, linked list (we purposely avoid this one in this book), radix tree, range tree, skip list, treap, etc.

The topic of Network Flow is much bigger than what we have wrote in Section 4.6 and the several sections in this chapter. Other topics like the Baseball Elimination problem, Circulation problem, Gomory-Hu tree, Push-relabel algorithm, Stoer-Wagner's min cut algorithm, and the rarely known Suurballe's algorithm can be added.

We can add more detailed discussions on a few more algorithms in Section 9.10, namely: Edmonds's Matching algorithm [13], Gale Shapley's algorithm for Stable Marriage problem, and Kuhn Munkres's (Hungarian) algorithm [39, 45].

There are many other mathematics problems and algorithms that can be added, e.g. the Chinese Remainder Theorem, modular multiplicative inverse, Möbius function, several exotic Number Theoretic problems, various numerical methods, etc.

In Section 6.4 and in Section 6.6, we have seen the KMP and Suffix Tree/Array solutions for the String Matching problem. String Matching is a well studied topic and other algorithms exist, like Aho Corasick's, Boyer Moore's, and Rabin Karp's.

In Section 8.2, we have seen several more advanced search techniques. Some programming contest problems are NP-hard (or NP-complete) problems but with small input size. The solution for these problems is usually a creative complete search. We have discussed several NP-hard/NP-complete problems in this book, but we can add more, e.g. Graph Coloring problem, Max Clique problem, Traveling Purchaser problem, etc.

Finally, we list down many other potential topic keywords that can possibly be included in the future editions of this book in alphabetical order, e.g. Burrows-Wheeler Transformation, Chu-Liu Edmonds's Algorithm, Huffman Coding, Karp's minimum mean-weight cycle algorithm, Linear Programming techniques, Malfatti circles, Min Circle Cover problem, Min Diameter Spanning Tree, Min Spanning Tree with one vertex with degree constraint, other computational geometry libraries that are not covered in Chapter 7, Optimal Binary Search Tree to illustrate the Knuth-Yao DP speedup [2], Rotating Calipers algorithm, Shortest Common Superstring problem, Steiner Tree problem, ternary search, Triomino puzzle, etc.

Statistics	First Edition	Second Edition	Third Edition
Number of Pages	-	-	58
Written Exercises	-	-	15*
Programming Exercises	-	-	80

Appendix A

uHunt

uHunt (`http://uhunt.felix-halim.net`) is a self-learning tool for UVa online-judge (UVa OJ [47]) created by one of the authors of this book (Felix Halim). The goal is to make solving problems at UVa OJ fun. It achieves the goal by providing:

1. Near real-time feedback and statistics on the recently submitted solutions so that the users can quickly iterate on improving their solutions (see Figure A.1). The users can immediately see the rank of their solutions compared to others in terms of performance. A (wide) gap between the user's solution performance with the best implies that the user still does not know a certain algorithms, data structures, or hacking tricks to get that faster performance. uHunt also has the 'statistics comparer' feature. If you have a *rival* (or a better UVa user that you admire), you can compare your list of solved problems with him/her and then try to solve the problems that your rival can solve.
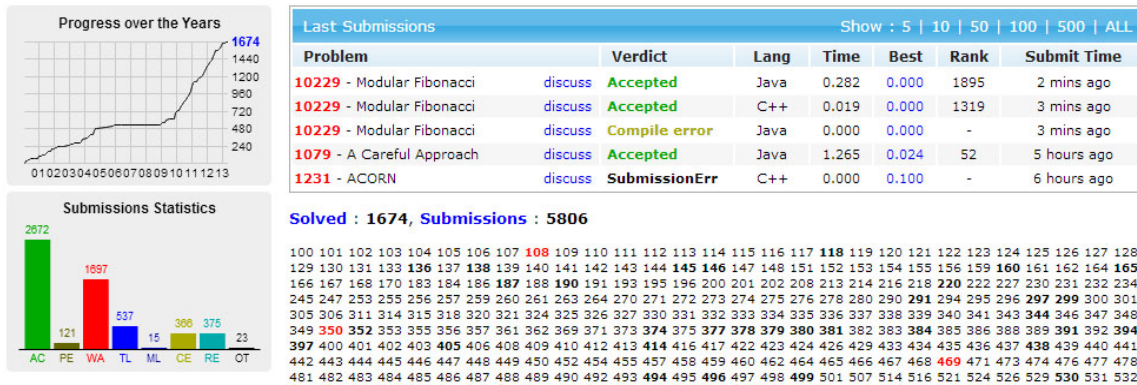
Figure A.1: Steven's statistics as of 24 May 2013

2. Web APIs for other developers to build their own tool. uHunt API has been used to create a full blown contest management system, a command line tool to submit solutions and get feedback through console, and mobile application to see the statistics.

3. A way for the users to help each others. The chat widget on the upper right corner of the page has been used to exchanges ideas and to help each other to solve problems. This gives a conducive environment for learning where user can always ask for help.

4. A selection of the next problems to solve, ordered by increasing difficulty (approximated by the number of distinct accepted users for the problems). This is useful for users who want to solve problems which difficulty matches their current skills. The rationale is this: If a user is still a beginner and he/she needs to build up his/her

confidence, he/she needs to solve problems with gradual difficulty. This is much better than directly attempting hard problems and keep getting non Accepted (AC) responses without knowing what's wrong. The ≈ 149008 UVa users actually contribute statistical information for each problem that can be exploited for this purpose. The easier problems will have higher number of submissions and higher number of AC. However, as a UVa user can still submit codes to a problem even though he/she already gets AC for that problem, then the number of AC alone is not an accurate measure to tell whether a problem is easy or not. An extreme example is like this: Suppose there is a hard problem that is attempted by a single good programmer who submits 50 AC codes just to improve his code's runtime. This problem is not easier than another easier problem where only 49 different users get AC. To deal with this, the default sorting criteria in uHunt is 'dacu' that stands for '*distinct* accepted users'. The hard problem in the extreme example above only has dacu = 1 whereas the easier problem has dacu = 49 (see Figure A.3).

Figure A.2: Hunting the next easiest problems using 'dacu'

5. A means to create virtual contests. Several users can decide to create a closed contest among them over a set of problems, with a certain contest duration. This is useful for team as well as individual training. Some contests have shadows (i.e. contestants from the past), so that the users can compare their skills to the real contestants in the past.

Figure A.3: We can rewind past contests with 'virtual contest'

6. An integration of ≈ 1675 programming exercises in this book from various categories (see Figure A.4). The users can keep track which programming exercises in this book that they have solved and see the progress of their work. These programming exercises can be used even without the book. Now, a user can customize his/her training programme to solve *problems of similar type*! Without such (manual) categorization, this training mode is hard to execute. We also give stars (*) to problems that we consider as **must try *** (up to 3 problems per category).

Figure A.4: The programming exercises in this book are integrated in uHunt

Building a web-based tool like uHunt is a computational challenge. There are over $\approx$ 11796315 submissions from $\approx$ 149008 users ($\approx$ one submission every few seconds). The statistics and rankings must be updated frequently and such update must be fast. To deal with this challenge, Felix uses lots of advanced data structures (some are beyond this book), e.g. database cracking [29], Fenwick Tree, data compression, etc.
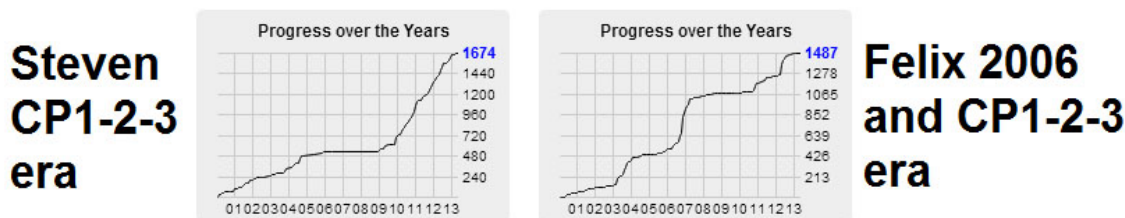
Figure A.5: Steven's & Felix's progress in UVa online judge (2000-present)

We ourselves are using this tool extensively in different stages of our life, as can be seen in Figure A.5. Two major milestones that can be seen from our progress chart are: Felix's intensive training to eventually won ACM ICPC Kaohsiung 2006 with his ICPC team (see Figure A.6) and Steven's intensive problem solving activities in the past four years (late 2009-present) to prepare this book.

Figure A.6: Andrian, Felix, and Andoko Won ACM ICPC Kaohsiung 2006

Appendix B

Credits

The problems discussed in this book are mainly taken from UVa online judge [47], ACM ICPC Live Archive [33], and past IOI tasks (mainly from 2009-2012). So far, we have contacted the following authors (and their current known affiliation as of 2013) to get their permissions (in alphabetical order):

1. Brian C. Dean (Clemson University, America)

2. Colin Tan Keng Yan (National University of Singapore, Singapore)

3. Derek Kisman (University of Waterloo, Canada)

4. Gordon V. Cormack (University of Waterloo, Canada)

5. Howard Cheng (University of Lethbridge, Canada)

6. Jane Alam Jan (Google)

7. Jim Knisely (Bob Jones University, America)

8. Jittat Fakcharoenphol (Kasetsart University, Thailand)

9. Manzurur Rahman Khan (Google)

10. Melvin Zhang Zhiyong (National University of Singapore, Singapore)

11. Michal (Misof) Forišek (Comenius University, Slovakia)

12. Mohammad Mahmudur Rahman (University of South Australia, Australia)

13. Norman Hugh Anderson (National University of Singapore, Singapore)

14. Ondřej Lhoták (University of Waterloo, Canada)

15. Petr Mitrichev (Google)

16. Piotr Rudnicki (University of Alberta, Canada)

17. Rob Kolstad (USA Computing Olympiad)

18. Rujia Liu (Tsinghua University, China)

19. Shahriar Manzoor (Southeast University, Bangladesh)

20. Sohel Hafiz (University of Texas at San Antonio, America)

21. Soo Yuen Jien (National University of Singapore, Singapore)

22. Tan Sun Teck (National University of Singapore, Singapore)

23. TopCoder, Inc (for PrimePairs problem in Section 4.7.4)

A compilation of photos with some of these problem authors that we managed to meet in person is shown below.

However, due to the fact that there are thousands (≈ 1675) of problems listed and discussed in this book, there are many problem authors that we have not manage to contact yet. If you are those problem authors or know the person whose problems are used in this book, please notify us. We keep a more updated copy of this problem credits in our supporting website: https://sites.google.com/site/stevenhalim/home/credits

Bibliography

[1] Ahmed Shamsul Arefin. *Art of Programming Contest (from Steven's old Website).* Gyankosh Prokashoni (Available Online), 2006.

[2] Wolfgang W. Bein, Mordecai J. Golin, Lawrence L. Larmore, and Yan Zhang. The Knuth-Yao Quadrangle-Inequality Speedup is a Consequence of Total-Monotonicity. *ACM Transactions on Algorithms*, 6 (1):17, 2009.

[3] Richard Peirce Brent. An Improved Monte Carlo Factorization Algorithm. *BIT Numerical Mathematics*, 20 (2):176–184, 1980.

[4] Brilliant. Brilliant.
https://brilliant.org/.

[5] Frank Carrano. *Data Abstraction and Problem Solving with C++: Walls and Mirrors.* Addison Wesley, 5th edition, 2006.

[6] Yoeng-jin Chu and Tseng-hong Liu. On the Shortest Arborescence of a Directed Graph. *Science Sinica*, 14:1396–1400, 1965.

[7] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Cliff Stein. *Introduction to Algorithm.* MIT Press, 2nd edition, 2001.

[8] Sanjoy Dasgupta, Christos Papadimitriou, and Umesh Vazirani. *Algorithms.* McGraw Hill, 2008.

[9] Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Cheong Schwarzkopf. *Computational Geometry: Algorithms and Applications.* Springer, 2nd edition, 2000.

[10] Edsger Wybe Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[11] Yefim Dinitz. Algorithm for solution of a problem of maximum flow in a network with power estimation. *Doklady Akademii nauk SSSR*, 11:1277–1280, 1970.

[12] Adam Drozdek. *Data structures and algorithms in Java.* Cengage Learning, 3rd edition, 2008.

[13] Jack Edmonds. Paths, trees, and flowers. *Canadian Journal on Maths*, 17:449–467, 1965.

[14] Jack Edmonds and Richard Manning Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of the ACM*, 19 (2):248–264, 1972.

[15] Susanna S. Epp. *Discrete Mathematics with Applications.* Brooks-Cole, 4th edition, 2010.

[16] Fabian Ernst, Jeroen Moelands, and Seppo Pieterse. Teamwork in Prog Contests: 3 * 1 = 4.
http://xrds.acm.org/article.cfm?aid=332139.

[17] Project Euler. Project Euler.
http://projecteuler.net/.

[18] Peter M. Fenwick. A new data structure for cumulative frequency tables. *Software: Practice and Experience*, 24 (3):327–336, 1994.

[19] Robert W. Floyd. Algorithm 97: Shortest Path. *Communications of the ACM*, 5 (6):345, 1962.

[20] Michal Forišek. IOI Syllabus.
http://people.ksp.sk/∼misof/ioi-syllabus/ioi-syllabus-2009.pdf.

[21] Michal Forišek. The difficulty of programming contests increases. In *International Conference on Informatics in Secondary Schools*, 2010.

[22] William Henry. Gates and Christos Papadimitriou. Bounds for Sorting by Prefix Reversal. *Discrete Mathematics*, 27:47–57, 1979.

[23] Felix Halim, Roland Hock Chuan Yap, and Yongzheng Wu. A MapReduce-Based Maximum-Flow Algorithm for Large Small-World Network Graphs. In *ICDCS*, 2011.

[24] Steven Halim and Felix Halim. Competitive Programming in National University of Singapore. In *A new learning paradigm: competition supported by technology*. Ediciones Sello Editorial S.L., 2010.

[25] Steven Halim, Roland Hock Chuan Yap, and Felix Halim. Engineering SLS for the Low Autocorrelation Binary Sequence Problem. In *Constraint Programming*, pages 640–645, 2008.

[26] Steven Halim, Roland Hock Chuan Yap, and Hoong Chuin Lau. An Integrated White+Black Box Approach for Designing & Tuning SLS. In *Constraint Programming*, pages 332–347, 2007.

[27] Steven Halim, Koh Zi Chun, Loh Victor Bo Huai, and Felix Halim. Learning Algorithms with Unified and Interactive Visualization. *Olympiad in Informatics*, 6:53–68, 2012.

[28] John Edward Hopcroft and Richard Manning Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM Journal on Computing*, 2 (4):225–231, 1973.

[29] Stratos Idreos. *Database Cracking: Towards Auto-tuning Database Kernels*. PhD thesis, CWI and University of Amsterdam, 2010.

[30] TopCoder Inc. Algorithm Tutorials.
http://www.topcoder.com/tc?d1=tutorials&d2=alg_index&module=Static.

[31] TopCoder Inc. PrimePairs. Copyright 2009 TopCoder, Inc. All rights reserved.
http://www.topcoder.com/stat?c=problem_statement&pm=10187&rd=13742.

[32] TopCoder Inc. Single Round Match (SRM).
http://www.topcoder.com/tc.

[33] Competitive Learning Institute. ACM ICPC Live Archive.
http://livearchive.onlinejudge.org/.

[34] IOI. International Olympiad in Informatics.
http://ioinformatics.org.

[35] Giuseppe F. Italiano, Luigi Laura, and Federico Santaroni. Finding Strong Bridges and Strong Articulation Points in Linear Time. *Combinatorial Optimization and Applications*, 6508:157–169, 2010.

[36] Arthur B. Kahn. Topological sorting of large networks. *Communications of the ACM*, 5 (11):558562, 1962.

[37] Juha Kärkkäinen, Giovanni Manzini, and Simon J. Puglisi. Permuted Longest-Common-Prefix Array. In *CPM, LNCS 5577*, pages 181–192, 2009.

[38] Jon Kleinberg and Eva Tardos. *Algorithm Design*. Addison Wesley, 2006.

[39] Harold W. Kuhn. The Hungarian Method for the assignment problem. *Naval Research Logistics Quarterly*, 2:83–97, 1955.

[40] Anany Levitin. *Introduction to The Design & Analysis of Algorithms*. Addison Wesley, 2002.

[41] Rujia Liu. *Algorithm Contests for Beginners (In Chinese)*. Tsinghua University Press, 2009.

[42] Rujia Liu and Liang Huang. *The Art of Algorithms and Programming Contests (In Chinese)*. Tsinghua University Press, 2003.

[43] Udi Manbers and Gene Myers. Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing*, 22 (5):935–948, 1993.

[44] Gary Lee Miller. Riemann's Hypothesis and Tests for Primality. *Journal of Computer and System Sciences*, 13 (3):300–317, 1976.

[45] James Munkres. Algorithms for the Assignment and Transportation Problems. *Journal of the Society for Industrial and Applied Mathematics*, 5(1):32–38, 1957.

[46] Institute of Mathematics and Lithuania Informatics. Olympiads in Informatics.
http://www.mii.lt/olympiads_in_informatics/.

[47] University of Valladolid. Online Judge.
http://uva.onlinejudge.org.

[48] USA Computing Olympiad. USACO Training Program Gateway.
http://train.usaco.org/usacogate.

[49] Joseph O'Rourke. *Art Gallery Theorems and Algorithms*. Oxford University Press, 1987.

[50] Joseph O'Rourke. *Computational Geometry in C*. Cambridge University Press, 2nd edition, 1998.

[51] David Pearson. A polynomial-time algorithm for the change-making problem. *Operations Research Letters*, 33 (3):231–234, 2004.

[52] John M. Pollard. A Monte Carlo Method for Factorization. *BIT Numerical Mathematics*, 15 (3):331–334, 1975.

[53] George Pólya. *How to Solve It*. Princeton University Press, 2nd edition, 1957.

[54] Janet Prichard and Frank Carrano. *Data Abstraction and Problem Solving with Java: Walls and Mirrors*. Addison Wesley, 3rd edition, 2010.

[55] Michael Oser Rabin. Probabilistic algorithm for testing primality. *Journal of Number Theory*, 12 (1):128–138, 1980.

[56] Kenneth H. Rosen. *Elementary Number Theory and its Applications*. Addison Wesley Longman, 4th edition, 2000.

[57] Kenneth H. Rosen. *Discrete Mathematics and its Applications*. McGraw-Hill, 7th edition, 2012.

[58] Robert Sedgewick. *Algorithms in C++, Part 1-5*. Addison Wesley, 3rd edition, 2002.

[59] Steven S. Skiena. *The Algorithm Design Manual*. Springer, 2008.

[60] Steven S. Skiena and Miguel A. Revilla. *Programming Challenges*. Springer, 2003.

[61] SPOJ. Sphere Online Judge.
http://www.spoj.pl/.

[62] Wing-Kin Sung. *Algorithms in Bioinformatics: A Practical Introduction*. CRC Press (Taylor & Francis Group), 1st edition, 2010.

[63] Robert Endre Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1 (2):146–160, 1972.

[64] Jeffrey Trevers and Stanley Milgram. An Experimental Study of the Small World Problem. *Sociometry*, 32 (4):425–443, 1969.

[65] Esko Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14 (3):249–260, 1995.

[66] Baylor University. ACM International Collegiate Programming Contest.
http://icpc.baylor.edu/icpc.

[67] Tom Verhoeff. 20 Years of IOI Competition Tasks. *Olympiads in Informatics*, 3:149166, 2009.

[68] Adrian Vladu and Cosmin Negruşeri. Suffix arrays - a programming contest approach. In *GInfo*, 2005.

[69] Henry S. Warren. *Hacker's Delight*. Pearson, 1st edition, 2003.

[70] Stephen Warshall. A theorem on Boolean matrices. *Journal of the ACM*, 9 (1):11–12, 1962.

[71] Wikipedia. The Free Encyclopedia.
http://en.wikipedia.org.

Index