Contents

Foreword

Long time ago (exactly the Tuesday November 11th 2003 at 3:55:57 UTC), I received an e-mail with the following sentence: 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" [34], 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 it's clear that was an exaggeration, to tell the truth I started thinking a bit about and I had a dream: to create a community around the project I had started as a part of my teaching job at UVa, with persons from everywhere around the world to work together after that ideal. Just by searching in Internet I immediately found a lot of people who was already creating a web-ring of sites with excellent tools to cover the many lacks of the UVa site.

The more impressive to me was the 'Methods to Solve' from Steven Halim, a very young student from Indonesia and I started to believe that the dream would become real a day, because the contents of the site were the result of a hard work of a genius of algorithms and informatics. Moreover his declared objectives matched the main part of my dream: to serve the humanity. And the best of the best, he has a brother with similar interest and capabilities, Felix Halim.

It's a pity it takes so many time to start a real collaboration, but the life is as it is. Fortunately, all of us have continued working in a parallel way and the book that you have in your hands is the best proof.

I can't imagine a better complement for the UVa Online Judge site, as it uses lots of examples from there carefully selected and categorized both by problem type and solving techniques, an incredible useful help for the users of the site. By mastering and practicing most programming exercises in this book, reader can easily go to 500 problems solved in UVa online judge, which will place them in top 400-500 within ≈100000 UVa OJ users.

Then it's clear that the book "Competitive Programming: Increasing the Lower Bound of Programming Contests" is suitable for programmers who wants 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 the newcomers, because 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 codes to implement the given algorithms. Because understand the problems is a thing, knowing the algorithms is another, and implementing them well in short and efficient code is tricky. After you read this extraordinary book three times you will realize that you are a much better programmer and, more important, a more happy person.

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 to master during their middle phase of their programming career if they wish to take a leap forward from being just another ordinary coder to being among one of the top finest programmer in the world.

Typical readers of this book would be:

1. University students who are competing in the annual ACM International Collegiate Programming Contest (ICPC) [38] Regional Contests (including the World Finals),
2. Secondary or High School Students who are competing in the annual International Olympiad in Informatics (IOI) [21] (including the National level),
3. Coaches who are looking for comprehensive training material for their students [13],
4. Anyone who loves solving problems through computer programs. There are numerous programming contests for those who are no longer eligible for ICPC like TopCoder Open, Google CodeJam, Internet Problem Solving Contest (IPSC), etc.

Prerequisites

This book is *not* written for novice programmers. When we wrote this book, we set it for readers who have basic knowledge in basic programming methodology, familiar with at least one of these programming languages (C/C++ or Java, preferably both), and have passed basic data structures and algorithms course typically taught in year one of Computer Science university curriculum.

Specific to the ACM ICPC Contestants

We know that one cannot probably win the ACM ICPC regional just by mastering the *current version (2nd edition)* of this book. While we have included a lot of materials in this book, we are much aware that much more than what this book can offer, are required to achieve that feat. Some reference pointers are listed in the chapter notes for those who are hungry for more. We believe, however, that your team would fare much better in future ICPCs after mastering this book.

Specific to the IOI Contestants

Same preface as above but with this additional Table 1. This table shows a list of topics that are currently not *yet* included in the IOI syllabus [10]. You can skip these items until you enter university (and join that university's ACM ICPC teams). However, learning them in advance may be beneficial as some harder tasks in IOI may require some of these knowledge.

Topic	In This Book
Data Structures: Union-Find Disjoint Sets	Section 2.3.2
Graph: Finding SCCs, Max Flow, Bipartite Graph	Section 4.2.1, 4.6.3, 4.7.4
Math: BigInteger, Probability, Nim Games, Matrix Power	Section 5.3, 5.6, 5.8, 5.9
String Processing: Suffix Tree/Array	Section 6.6
More Advanced Topics: A*/IDA*	Section 8.3

Table 1: Not in IOI Syllabus [10] *Yet*

We know that one cannot win a medal in IOI just by mastering the *current version* of this book. While we believe many parts of the IOI syllabus have been included in this book – which should give you a respectable score in future IOIs – we are well aware that modern IOI tasks requires more problem solving skills and creativity that we cannot teach via this book. So, keep practicing!

Specific to the Teachers/Coaches

This book is used in Steven's CS3233 - 'Competitive Programming' course in the School of Computing, National University of Singapore. It 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. Hints/brief solutions of the written exercises in this book are given in Appendix A. Fellow teachers/coaches are free to modify the lesson plan to suit your students' needs.

Wk	Topic	In This Book
01	Introduction	Chapter 1
02	Data Structures & Libraries	Chapter 2
03	Complete Search, Divide & Conquer, Greedy	Section 3.2-3.4
04	Dynamic Programming 1 (Basic Ideas)	Section 3.5
05	Graph 1 (DFS/BFS/MST)	Chapter 4 up to Section 4.3
06	Graph 2 (Shortest Paths; DAG-Tree)	Section 4.4-4.5; 4.7.1-4.7.2
-	Mid semester break	-
07	Mid semester team contest	-
08	Dynamic Programming 2 (More Techniques)	Section 6.5; 8.4
09	Graph 3 (Max Flow; Bipartite Graph)	Section 4.6.3; 4.7.4
10	Mathematics (Overview)	Chapter 5
11	String Processing (Basic skills, Suffix Array)	Chapter 6
12	(Computational) Geometry (Libraries)	Chapter 7
13	Final team contest	All, including Chapter 8
-	No final exam	-

Table 2: Lesson Plan

To All Readers

Due to the diversity of its content, this book is *not* meant to be read once, but several times. There are many written exercises and programming problems ($\approx$ 1198) scattered throughout the body text of this book which can be skipped at first if the solution is not known at that point of time, but can be revisited later after the reader has accumulated new knowledge to solve it. Solving these exercises will strengthen the concepts taught in this book as they usually contain interesting twists or variants of the topic being discussed. Make sure to attempt them once.

We believe this book is and will be relevant to many university and high school students as ICPC and IOI will be around for many years ahead. New students will require the 'basic' knowledge presented in this book before hunting for more challenges after mastering this book. But before you assume anything, please check this book's table of contents to see what we mean by 'basic'.

We will be happy if in the year 2010 (the publication year of the first edition of this book) and beyond, the *lower bound* level of competitions in ICPC and IOI increases because many of the contestants have mastered the content of this book. That is, we want to see fewer teams solving very few problems (≤ 2) in future ICPCs and fewer contestants obtaining *less than* 200 marks in future IOIs. We also hope to see many ICPC and IOI coaches around the world, especially in South East Asia, adopt this book knowing that without mastering the topics *in and beyond* this book, their students have very little chance of doing well in future ICPCs and IOIs. If such increase in 'required lowerbound knowledge' happens, this book would have fulfilled its objective of advancing the level of human knowledge in this era.

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 several feedbacks from the readers, especially from Steven's CS3233 class Sem 2 AY2010/2011 students, that we have incorporated 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 more information density per page. The 2nd edition uses single line spacing instead of one half line spacing 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 we add more content.
- Some minor bug in our example codes (both the ones shown in the book and the soft copy given in the companion web site) are fixed. All example codes now have much more meaningful comments to help readers understand the code.
- Several known language issues (typo, grammatical, stylistic) have been corrected.
- On top of enhancing the writing of existing data structures, algorithms, and programming problems, we also add these *new* materials in each chapter:

  1. Much more Ad Hoc problems to kick start this book (Section 1.3).
  2. Lightweight set of Boolean (bit manipulation techniques) (Section 2.2.1), Implicit Graph (Section 2.3.1), and Fenwick Tree data structure (Section 2.3.4).
  3. More DP: Clearer explanation of the bottom-up DP, $O(n \log k)$ solution for LIS problem, Introducing: 0-1 Knapsack/Subset Sum, DP TSP (bitmask technique) (Section 3.5.2).
  4. Reorganization of 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: Prim's MST algorithm, Explaining DP as a traversal on implicit DAG (Section 4.7.1), Eulerian Graph (Section 4.7.3), Alternating Path Bipartite Matching algorithm (Section 4.7.4).
  5. Reorganization of mathematics topics (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 is rewritten and made clearer.
  6. Basic string processing skills (Section 6.2), more string problems (Section 6.3), string matching (Section 6.4), and an enhanced Suffix Tree/Array explanation (Section 6.6).
  7. Much more geometric libraries (Chapter 7), especially on points, lines, and polygons.
  8. New Chapter 8: Discussion on problem decomposition; More advanced search techniques (A*, Depth Limited Search, Iterative Deepening, IDA*); More advanced DP: more bitmasks techniques, Chinese Postman Problem, compilation of common DP states, discussion on better DP states, and some other harder DP problems.

- Many existing figures in this book are re-drawn and enhanced. Many new figures are added to help explain the concepts more clearly.
- The first edition is mainly written using ICPC and C++ viewpoint. The second edition is now written in a more balanced viewpoint between ICPC versus IOI and C++ versus Java. Java support is strongly enhanced in the second edition. However, we do not support any other programming languages as of now.
- Steven's 'Methods to Solve' website is now fully integrated in this book in form of 'one liner hints' per 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 now consider that this feat is doable by a *serious* 4-year Computer Science university undergraduate).
- Some examples in the first edition use old programming problems. In the second edition, some examples are replaced/added with the newer examples.
- Addition of around 600 more programming exercises that Steven & Felix have solved in UVa online judge and Live Archive between the first and second edition. We also give much more conceptual exercises throughout the book with hints/short solutions as appendix.
- Addition of short profiles of data structure/algorithm inventors adapted from Wikipedia [42] or other sources. It is nice to know a little bit more about the man behind these algorithms.

Web Sites

This book has an official companion web site at: `http://sites.google.com/site/stevenhalim`. In that website, you can download the soft copy of sample source codes and PDF slides (but only the *public/simpler version*) used in Steven's CS3233 classes.

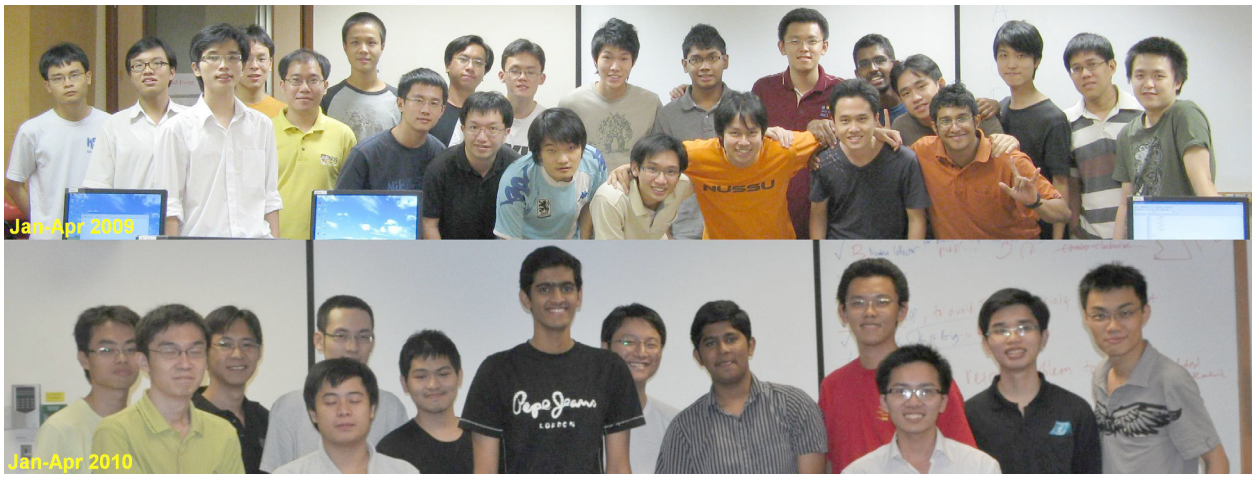
All programming exercises in this book are integrated in:
`http://felix-halim.net/uva/hunting.php`, and in
`http://uva.onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&category=118`

Acknowledgments for the First Edition

Steven wants to thank:

- God, Jesus Christ, Holy Spirit, for giving talent and passion in this 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.
- School of Computing, National University of Singapore, for employing me and allowing me to teach CS3233 - 'Competitive Programming' module from which this book is born.
- NUS/ex-NUS professors/lecturers who have shaped my competitive programming and coaching skills: Prof Andrew Lim Leong Chye, Dr Tan Sun Teck, Aaron Tan Tuck Choy, Dr 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.

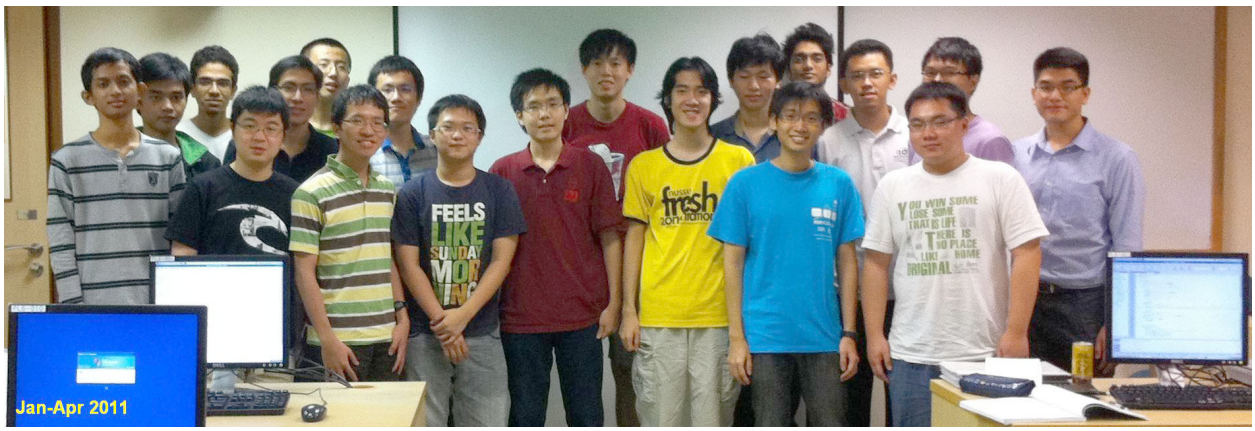
- My CS3233 students in Sem2 AY2008/2009 who inspired me to come up with the lecture notes and students in Sem2 AY2009/2010 who verified the content of the first edition of this book and gave the initial Live Archive contribution



Acknowledgments for the Second Edition

Additionally, Steven wants to thank:

- ≈ 550 buyers of the 1st edition as of 1 August 2011. Your supportive responses encourage us!
- 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 of this book, 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 (class photo is shown below).



- The proof readers: Seven of CS3233 students above (underlined) plus Tay Wenbin.
- Last but not least, Steven wants to re-thank his wife, Grace Suryani, for letting me do another round of tedious book editing process while you are pregnant with our first baby.

To a better future of humankind,
Steven and Felix Halim
Singapore, 1 August 2011

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 up to the 'Competitive Programming' module that uses this book. He is the coach of both NUS ACM ICPC teams and 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 one ACM ICPC World Finalist team (2009-2010) as well as two gold, four silver, and six bronze IOI medallists (2009-2011).

As seen from the family photo, Steven is a happily married man. His wife, Grace Suryani, is currently pregnant with our first baby during the time the second edition of this book is released.

Felix Halim, PhD Candidate[2]

felix.halim@gmail.com

Felix Halim is currently a PhD student in the same University: SoC, NUS. In terms of programming contests, Felix has a much more colorful 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 Finalist @ Tokyo 2007 (Honorable Mention). Today, he actively joins TopCoder Single Round Matches and his highest rating is a **yellow** coder.

Copyright

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

[2]Research area: "Large Scale Data Processing".

Convention

There are a lot of C++ codes shown in this book. If they appear, they will be written `using this font`. Many of them use typedefs, shortcuts, or macros that are commonly used by competitive programmers to speed up the coding time. In this short section, we list down several examples. Java support has been increased substantially in the second edition of this book. This book uses Java which, as of now, does not support macros and typedefs.

```
// 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 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

// Note that we abandon the usage of "REP" and "TRvii" in the second edition
// to reduce the confusion encountered by new programmers
```

The following shortcuts are frequently used in our C/C++/Java codes in this book:

```
// ans = a ? b : c;                // to simplify: if (a) ans = b; else ans = c;
// index = (index + 1) % n;         // from: index++; if (index >= n) index = 0;
// index = (index + n - 1) % n;  // from: index--; if (index < 0) index = n - 1;
// int ans = (int)((double)d + 0.5);          // for rounding to nearest integer
// ans = min(ans, new_computation)    // we frequently use this min/max shortcut
// some codes uses short circuit && (AND) and || (OR)
```

Problem Categorization

As of 1 August 2011, Steven and Felix – combined – have solved 1502 UVa problems ($\approx 51\%$ of the entire UVa problems). About ≈ 1198 of them are discussed and categorized in this book.

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 problems 'wrongly' categorized or problems whose category does not match the technique you use to solve it. What we can guarantee is this: If you see problem X in category Y, then you know that *we* have solved problem X with the technique mentioned in the section that discusses category Y.

If you need hints for any of the problems, you may turn to the index at the back of this book and save yourself the time needed to flip through the whole book to understand any of the problems. The index contains a sorted list of UVa/LA problems number (do a binary search!) which will help locate the pages that contains the discussion of those problems (and the required data structures and/or algorithms to solve that problem).

Utilize this categorization feature for your training! To diversify your problem solving skill, it is a good idea to solve at least few problems from each category, especially the ones that we highlight as **must try *** (we limit ourself to choose maximum 3 highlights per category).

Abbreviations

A* : A Star
ACM : Association 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

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

ED : Edit Distance

FT : Fenwick Tree

GCD : Greatest Common Divisor

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

LA : Live Archive [20]
LCA : Lowest Common Ancestor
LCM : Least Common Multiple
LCP : Longest Common Prefix
LCS_1 : Longest Common Subsequence
LCS_2 : Longest Common Substring
LIS : Longest Increasing Subsequence
LRS : Longest Repeated Substring

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
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 Minimum (or Maximum) 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 [28]

WA : Wrong Answer
WF : World Finals

List of Tables

List of Figures

Chapter 1

Introduction

I want to compete in ACM ICPC World Finals!
— A dedicated student

1.1 Competitive Programming

'Competitive Programming' in summary, 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). Definitely, some people (at least the problem author) have solved these problems before. 'Solve them' implies that we[1] must push our CS knowledge to a certain required level so that we can produce working codes that can solve these problems too – in terms of getting the *same* output as the problem setter using the problem setter's secret[2] test data. 'As quickly as possible' is the competitive element which is a very natural human behavior.

Please note that being well-versed in competitive programming is *not* the end goal. It is just the means. The true end goal is to produce all-rounder computer scientists/programmers who are much more ready to produce better software or to face harder CS research problems in the future. The founders of ACM International Collegiate Programming Contest (ICPC) [38] have this vision and we, the authors, agree with it. With this book, we play our little roles in preparing current and future generations to be more competitive in dealing with well-known CS problems frequently posed in the recent ICPCs and International Olympiad in Informatics (IOI)s.

Illustration on solving UVa Online Judge [28] Problem Number 10911 (Forming Quiz Teams).

Abridged Problem Description:

Let (x, y) be the coordinate 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: $N \leq 8; 0 \leq x, y \leq 1000$.

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

Sample output:
$cost = 4.83$.

Think first, try not to flip this page immediately!

---

[1]Some programming competitions are done in 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 possible corner cases of the problem and test their program with those cases. This is typical in real life when software engineers have to test their software a lot to make sure the software met the requirements set by the customers.

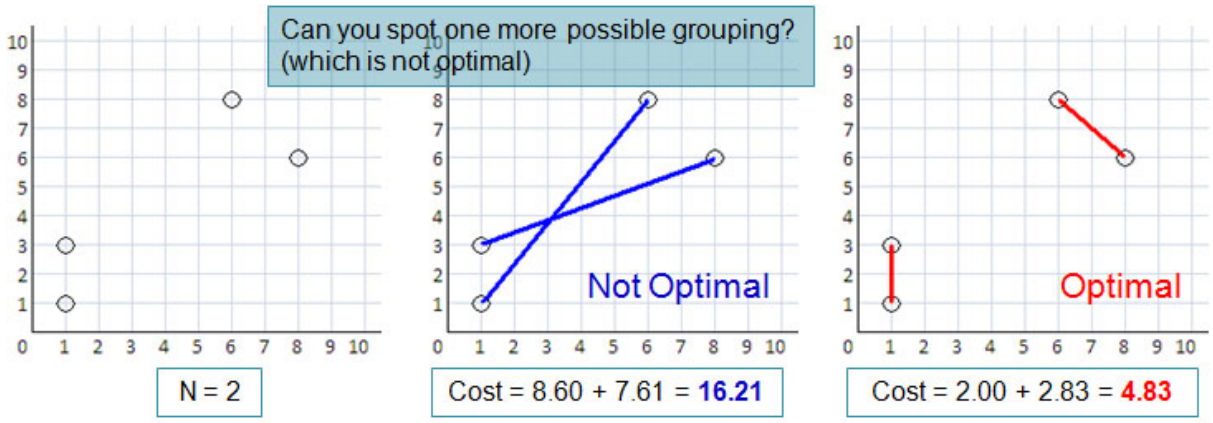


Figure 1.1: Illustration of UVa 10911 - Forming Quiz Teams

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

- Non-competitive programmer A (a.k.a the blurry one):
  Step 1: He reads the problem and confused; He has never seen this kind of problem before.
  Step 2: He tries to code something, starting from reading the non-trivial input and output.
  Step 3: He then realizes that all his attempts fail:
  Greedy solution: Pair students based on shortest distances gives **Wrong Answer (WA)**.
  Complete search using backtracking gives **Time Limit Exceeded (TLE)**.
  After 5 hours of labor (typical contest time), no **Accepted (AC)** solution is produced.
- Non-competitive programmer B (Give up):
  Step 1: Read the problem...
  Then realize that he has seen this kind of problem before.
  But also remember that he has not learned how to solve this kind of problem...
  He is not aware of a simple solution for this problem: **Dynamic Programming (DP)**...
  Step 2: Skip the problem and read another problem in the problem set.
- (Still) non-competitive programmer C (Slow):
  Step 1: Read the problem and realize that it is a **'matching on general graph'** problem.
  In general, this problem **must** be solved using **'Edmonds' Blossom'** algorithm [6].
  But 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.
  Step 2: Code I/O routine, write recursive top-down DP, test the solution, **debug** >.<...
  Step 3: *Only after 3 hours*, his solution is judged as AC (passed all secret test data).
- Competitive programmer D:
  Same as programmer C, but completing all the steps above in less than 30 minutes.
- Very Competitive programmer E:
  A very competitive programmer (e.g. the red 'target' coders in TopCoder [19]) should solve this 'well known' problem in less than 15 minutes...

1.2 Tips to be Competitive

If you strive to be like competitive programmers D or E above – that is, if you want to do well to qualify and get a medal in IOI [21], or to qualify for ACM ICPC [38] nationals, regionals, and up to the world finals, or in other programming contests – then this book is definitely for you!

In the subsequent chapters, you will learn basic to more intermediate/advanced data structures and algorithms frequently appearing in recent programming contests, compiled from many sources [30, 5, 31, 3, 24, 32, 26, 34, 1, 23, 4, 33, 25, 36, 27] (see Figure 1.4).

However, you will not just learn 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 mainly taken from our own experience that can be helpful in contest situations. We start this book by giving you several general tips below:

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 recent ICPCs where rank i and rank $i+1$ are just separated by few minutes. When you can solve the same number of problems as your competitor, it is now down to coding skill (ability to produce shorter + correct 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. You also need to familiarize your fingers with the positions of the frequently used programming language characters, e.g. braces {} or () or <>, semicolon ';', single quote for 'char', double quotes for "string", etc.

As a little practice, try typing this C++ code as fast as possible.

```
/* Forming Quiz Teams, the solution for UVa 10911 above */

#include <algorithm>   // note: if you have problems understanding this C++ code
#include <cmath>                                          // (not the algorithm)
#include <cstdio>   // please consult your basic programming text books first...
#include <cstring>       // the explanation of this Dynamic Programming solution
using namespace std;           // can be found in Chapter 3 and Chapter 8 later.
                                 // See Section 2.2.1 for this bitmask technique
          // using global variables is not a good software engineering practice,
int N;                                  // but it is OK for competitive programming
double dist[20][20], memo[1 << 16];    // 1 << 16 is 2^16, recall that max N = 8

double matching(int bitmask) {                              // DP state = bitmask
  // see that we initialize the array 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 == (1 << (2 * N)) - 1)                  // all are already matched
    return memo[bitmask] = 0;                                    // cost is 0

  double ans = 2000000000.0;                    // initialize with a large value
  for (int p1 = 0; p1 < 2 * N; p1++)
    if (!(bitmask & (1 << p1))) {                             // if this bit is off
      for (int p2 = p1 + 1; p2 < 2 * N; p2++)
        if (!(bitmask & (1 << p2)))         // if this different bit is also off
          ans = min(ans,
                    dist[p1][p2] + matching(bitmask | (1 << p1) | (1 << p2)));
      break;         // important step, will be discussed in Chapter 3 & 8 later
    }

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

int main() {
  char line[1000], name[1000];
  int i, j, caseNo = 1, x[20], y[20];
  // to simplify testing, redirect input file to stdin
  // freopen("10911.txt", "r", stdin);
```

```
  // we use gets(line) to read input line by line
  while (sscanf(gets(line), "%d", &N), N) {            // yes, we can do this :)
    for (i = 0; i < 2 * N; i++)
      sscanf(gets(line), "%s %d %d", &name, &x[i], &y[i]);

    for (i = 0; i < 2 * N; i++)                 // build pairwise distance table
      for (j = 0; j < 2 * N; j++)               // have you used 'hypot' before?
        dist[i][j] = hypot((double)x[i] - x[j], (double)y[i] - y[j]);

    // use DP to solve minimum weighted perfect matching on small general graph
    // this is a trick to initialize the memo table with -1
    memset(memo, -1, sizeof memo);
    printf("Case %d: %.2lf\n", caseNo++, matching(0));
} } // return 0;
```

FYI, the explanation of this Dynamic Programming algorithm with bitmask technique is given in Section 2.2.1, 3.5, and 8.4.1. You do not have to be alarmed if you do not understand it yet.

1.2.1 Tip 2: Quickly Identify Problem Types

In ICPCs, the contestants (teams) are given a **set** of problems ($\approx$ 7-11 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 (2009-2010), the contestants are given 8 tasks over 2 days (6 tasks in 2011) that covers items no 1-5 and 10, plus a *much smaller* subset of items no 6-10 in Table 1.1. For details, please refer to the IOI syllabus 2009 [10] and the IOI 1989-2008 problem classification [39].

No	Category	In This Book	Appearance Frequency
1.	Ad Hoc	Section 1.3	1-2
2.	Complete Search (Iterative/Recursive)	Section 3.2	1-2
3.	Divide & Conquer	Section 3.3	0-1
4.	Greedy (only the original ones)	Section 3.4	0-1
5.	Dynamic Programming (only the original ones)	Section 3.5	1-2 (can go up to 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 Problems	Chapter 8	0-1
		Total in Set	7-15 (usually ≤ 11)

Table 1.1: Recent ACM ICPC Asia Regional Problem Types

The classification in Table 1.1 is adapted from [29] and by no means complete. Some problems, e.g. 'sorting', are not classified here as they are 'trivial' and only used as a 'sub-routine' in a bigger problem. We do not include 'recursion' as it is embedded in other categories. We also omit 'data structure' as usage of efficient data structure is integral for solving harder problems.

Of course there can be a mix and match of problem types: One problem can be classified into more than one type, e.g. Floyd Warshall's algorithm is either a solution for graph problem: All-Pairs Shortest Paths (APSP, Section 4.5) or a Dynamic Programming (DP) algorithm (Section 3.5). Prim's and Kruskal's algorithms are either the solutions for another graph problem: Minimum Spanning Tree (MST, Section 4.3) or Greedy algorithms (Section 3.4).

In the future, these classifications may change. One significant example is DP. This technique was not known before 1940s, not frequently used in ICPCs or IOIs before mid 1990s, but it is a must today. There are ≥ 3 DP problems (out of 11) in the recent ICPC World Finals 2010.

Exercise 1.2.1: Read the UVa [28] problems shown in Table 1.2 and determine their problem types. The first and the fifth ones have been filled-in for you. Filling this table is easy after mastering this book as all techniques to solve these problems are discussed in this book. Please revisit this exercise when you have finished reading this book once.

UVa	Title	Problem Type	Hint
10360	Rat Attack	Complete Search or Dynamic Programming	Section 3.2 or 3.5
10341	Solve It		Section 3.3
11292	Dragon of Loowater		Section 3.4
11450	Wedding Shopping		Section 3.5
10911	Forming Quiz Teams	DP + bitmask	Section 8.4.1
11635	Hotel Booking		Section 8.2
11506	Angry Programmer		Section 4.6
10243	Fire! Fire!! Fire!!!		Section 4.7.1
10717	Mint		Section 8.2
11512	GATTACA		Section 6.6
10065	Useless Tile Packers		Section 7.3.7

Table 1.2: Exercise: Classify These UVa Problems

The goal is not just to map problems into categories as in Table 1.1. After you are familiar with most of the topics in this book, you can classify the problems into just four types as in Table 1.3.

No	Category	Confidence and Expected Solving Speed
A.	I have solved this type before	I am confident that I can solve it again now (and fast)
B.	I have solved this type before	But I know coding the solution takes time
C.	I have seen this type before	But that time I have not or cannot solve it
D.	I have not seen this type before	I may (not) be able to solve it now (under contest time)

Table 1.3: Problem Types (Compact Form)

To be competitive, you must frequently classify the problems that you read in the problem set into type A (or at least type B).

1.2.2 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. However, some of them may be incorrect, some of them are not fast enough, and some of them are 'overkill'... The rule of thumb is: Brainstorm many possible algorithms - then pick the **simplest that works** (fast enough to pass the time and memory limit, yet still produce the correct answer)[3]!

For example, the maximum size of input n is $100K$, or 10^5 ($1K = 1,000$), and your algorithm has time complexity of order $O(n^2)$. Your common sense tells you that $(100K)^2$ is an extremely big number, it is 10^{10}. So, you will try to devise a faster (and correct) algorithm to solve the problem, say of order $O(n \log_2 n)$. Now $10^5 \log_2 10^5$ is just 1.7×10^6. Since computer nowadays are quite fast and can process up to $\approx 1M$, or 10^6 ($1M = 1,000,000$) operations in one second, your common sense tells you that this one is likely able to pass the time limit.

[3]Discussion: It is true that in programming contests, picking the simplest algorithm that works is crucial. However, during *training sessions*, where we are not under time constraint, it is beneficial if we spent more time trying to solve a certain problem using the *best possible algorithm*, because maybe in the future contests, we meet a harder version of the problem. If we are well prepared, the chance that we can come up with the correct solution is higher.

Now how about this scenario: You can only devise an algorithm of order $O(n^4)$. Seems pretty bad right? But if $n \leq 10$, then you are done. Just directly implement your $O(n^4)$ algorithm since 10^4 is just $10K$ and your algorithm will only use a relatively small computation time.

So, by analyzing the complexity of your algorithm with the given input bound and stated time/memory limit, you can judge whether you should try coding your algorithm (which will take your time, especially in the time-constrained ICPCs and IOIs), attempt to improve your algorithm first, or switch to other problems in the problem set.

In this book, we will *not* discuss the concept of algorithm analysis in details. We *assume* that you already have this basic skill. Please check other reference books (for example, the "Introduction to Algorithms" [3], "Algorithm Design" [23], etc) and make sure that you understand how to:

- Prove the correctness of an algorithm (especially for Greedy algorithms in Section 3.4).
- Analyze time/space complexity for iterative and recursive algorithms.
- Perform amortized analysis (see [3], Chapter 17) – although rarely used in contests.
- Do output-sensitive analysis, to analyze algorithm which depends on output size. Example: The $O(m + occ)$ complexity to find a pattern string with length m in a long string with length n with the help of Suffix Tree (see Section 6.6.3). See that the time complexity does not depend on the input size n but on the output size m and number of *occ*(urences).

Many novice programmers would skip this phase and attempt to directly code the first algorithm that they can think of (usually the naïve version) only to realize that the chosen data structure is not efficient or their algorithm is not fast enough (or wrong). Our advice for ICPC contestants[4]: Refrain from coding until you are sure that your algorithm is both correct and fast enough.

To help you in judging how fast is 'enough', we produce Table 1.4. Variants of such Table 1.4 can also be found in many algorithms book. However, we put another one here from a programming contest's perspective. Usually, the input size constraints are given in (good) problem description. Using some logical assumptions that typical year 2011 CPU can do $1M$ operations in 1s and time limit of 3s (typical time limit used in most UVa online judge [28] problems), we can predict the 'worst' algorithm that can still pass the time limit. Usually, the simplest algorithm has poor time complexity, but if it can pass the time limit, just use it!

n	Worst AC Algorithm	Comment
≤ 10	$O(n!), O(n^6)$	e.g. Enumerating a permutation (Section 3.2)
≤ 15	$O(2^n \times n^2)$	e.g. DP TSP (Section 3.5.2)
≤ 20	$O(2^n), O(n^5)$	e.g. DP + bitmask technique (Section 8.4.1)
≤ 50	$O(n^4)$	e.g. DP with 3 dimensions + $O(n)$ loop, choosing ${}_nC_{k=4}$
≤ 100	$O(n^3)$	e.g. Floyd Warshall's (Section 4.5)
$\leq 1K$	$O(n^2)$	e.g. Bubble/Selection/Insertion Sort (Section 2.2.1)
$\leq 100K$	$O(n \log_2 n)$	e.g. Merge Sort, building Segment Tree (Section 2.2.2)
$\leq 1M$	$O(n), O(\log_2 n), O(1)$	Usually, contest problem has $n \leq 1M$ (e.g. to read input)

Table 1.4: Rule of Thumb for the 'Worst AC Algorithm' for various input size n (single test case only), assuming that year 2011 CPU can compute $1M$ items in 1s and Time Limit of 3s.

From Table 1.4, we see the importance of knowing good algorithms with lower order of growth as they allow us to solve problems with bigger input size. Beware that a faster algorithm is usually non trivial and harder to code. In Section 3.2.2 later, we will see a few tips that may allow us to enlarge the possible input size n for the same class of algorithm. In the other chapters in this book, we will see efficient algorithms for various computing problems.

[4]Discussion: Unlike ICPC, the tasks presented in IOI usually can be solved with several possible solutions with different efficiency and thus different scoring criteria. To gain valuable points, it may be beneficial to initially use a brute force solution to score few points and to understand the problem better. There are no significant time penalty as IOI is not a speed contest. Then, iteratively improve the solution to gain more points.

Additionally, we have a few other rules of thumb that are useful in programming contests:

- $2^{10} = 1,024 \approx 10^3$, $2^{20} = 1,048,576 \approx 10^6$.
- Max 32-bit signed integer: $2^{31} - 1 \approx 2 \times 10^9$ (safe for up to ≈ 9 decimal digits);
  Max 64-bit signed integer (long long) is $2^{63} - 1 \approx 9 \times 10^{18}$ (safe for up to ≈ 18 decimal digits).
  Use 'unsigned' if slightly higher positive number is needed `[0..`$2^{64} - 1$`]`.
  If you need to store integers $\geq 2^{64}$, you need to use the Big Integer technique (Section 5.3).
- Program with nested loops of depth k running about n iterations each has $O(n^k)$ complexity.
- If your program is recursive with b recursive calls per level and has L levels, the program has roughly $O(b^L)$ complexity. But this is an upper bound. The actual complexity depends on what actions done per level and whether some pruning are possible.
- There are $n!$ permutations and 2^n subsets (or combinations) of n elements.
- Dynamic Programming algorithms which fill in a 2D matrix in $O(k)$ per cell is in $O(k \times n^2)$. More details in Section 3.5 later.
- The best time complexity of a comparison-based sorting algorithm is $\Omega(n \log_2 n)$.
- Most of the time, $O(n \log_2 n)$ algorithms will be sufficient for most contest problems.
- The largest input size for typical programming contest problems must be $<< 1M$, because beyond that, the time needed to read the input (the I/O routine) will be the bottleneck.

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 re-attempt this exercise again.

1. There are n webpages ($1 \leq n \leq 10M$). Each webpage i has different page rank r_i. You want to pick the top 10 pages with highest page ranks. Which method is more feasible?
   (a) Load all n webpages' page rank to memory, sort (Section 2.2.1), and pick the top 10.
   (b) Use priority queue data structure (heap) (Section 2.2.2).
2. Given a list L of up to $10K$ integers. You need to *frequently* ask the value of `sum(i, j)`, i.e. the sum of `L[i] + L[i+1] + ...+ L[j]`. Which data structure should you use?
   (a) Simple Array (Section 2.2.1).
   (b) Simple Array that is pre-processed with Dynamic Programming (Section 2.2.1 & 3.5).
   (c) Balanced Binary Search Tree (Section 2.2.2).
   (d) Hash Table (Section 2.2.2).
   (e) Segment Tree (Section 2.3.3).
   (f) Fenwick Tree (Section 2.3.4).
   (g) Suffix Tree (Section 6.6.2).
   (h) Suffix Array (Section 6.6.4).
3. Given a set S of N points *randomly* scattered on 2D plane, $N \leq 1000$. Find two points $\in S$ that has the greatest Euclidian distance. Is $O(N^2)$ complete search algorithm that try all possible pairs feasible?
   (a) Yes, such complete search is possible.
   (b) No, we must find another way.

4. 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 contest setting?
   (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).

5. Which algorithm is faster (based on its time complexity) for producing a list of the first $10K$ prime numbers? (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).

6. You want to test if factorial of n, i.e. $n!$ is divisible by an integer m. $1 \leq n \leq 10000$. What should you do?
   (a) Test if $n!$ % $m == 0$.
   (b) The naïve approach above will not work. We must use: _____ (Section 5.5.1).

7. You want to know all the occurrences of a substring P (of length m) in a (long) string T (of length n), if any. Both n and m can go up to 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 (P[j] != T[i + j]) found = false;
  if (found) printf("P = %s is found in index %d in T\n", P, i);
}
```

   (b) The naïve approach above will not work. We must use: _____ (Section 6.4).

8. Same problem as in question 3 earlier, but now the number of points N is larger: $N \leq 1M$. Recall that the points are *randomly scattered* on 2D plane.
   (a) The complete search mentioned in question 3 can still be used.
   (b) The naïve approach above will not work. We must use: _____ (Section 7.3.7).

1.2.3 Tip 4: Master Programming Languages

There are several programming languages allowed in ICPC[5], including C/C++ and Java. Which one should we master?

Our experience gave us this answer: Although we prefer C++ with built-in Standard Template Library (STL), we still need to master Java, albeit slower, since Java has powerful BigInteger/BigDecimal, String Processing, GregorianCalendar API, etc. Java is more debugging friendly as a Java program displays its 'stack trace' when it 'crash' (as opposed to C/C++ 'core dump').

[5]Important note: Looking at the IOI 2011 competition rules, Java is currently still not allowed in IOI. The programming languages allowed in IOI are C, C++, and Pascal. On the other hand, ICPC World Finals (and thus most Regionals) allows C, C++ and Java. Looking at this situation, it is seems that the 'best' language to master is C++ as it is allowed in both competitions and it has strong STL support.

A simple illustration is shown below (this is part of the solution for UVa problem 623: 500!): Compute 25! (factorial of 25). The answer is very large: 15,511,210,043,330,985,984,000,000. This is way beyond the largest built-in integer data type (`unsigned long long`: $2^{64}-1$) in C/C++. Using C/C++, you will have hard time coding this simple problem as there is no native support for BigInteger data structure in C/C++ yet. Meanwhile, the Java code is simply this:

```
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));                          // wow :)
    System.out.println(fac);
} }
```

Another illustration to further demonstrate that mastering a programming language is important: Please read this input: The first line of input is an integer N. Then, there are N lines that start with the character '0', followed by one dot '.', followed by an unknown number of digits (up to 100 digits), and finally terminated with three dots '...'. See the example below.

```
3
0.1227...
0.517611738...
0.7341231223444344389923899277...
```

One solution is as follows:

```
#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 slightly larger than needed

int main() {
  scanf("%d", &N);
  while (N--) {                         // we simply loop from N, N-1, N-2, ... 0
    scanf("0.%[0-9]...", &x);             // if you are surprised with this line,
                                    // check scanf details in www.cppreference.com
    printf("the digits are 0.%s\n", x);
  }
  return 0;
}
```

Example codes: `ch1_01_factorial.java`; `ch1_02_scanf.cpp`

Not many C/C++ programmers are aware of the trick above. Although `scanf/printf` are C-style I/O routines, they can still be used in a C++ code. Many C++ programmers 'force' themselves to use `cin/cout` all the time which, in our opinion, are not as flexible as `scanf/printf` and slower.

In programming contests, especially ICPCs, coding should *not* be your bottleneck at all. That is, once you figure out the 'worst AC algorithm' that will pass the given time limit, you are supposed to be able to translate it into a bug-free code and you can do it fast!

Now, try to do some exercises below! If you need more than 10 lines of code to solve them, you will need to relearn your programming language(s) in depth! Mastery of programming language routines will help you a lot in programming contests.

Exercise 1.2.3: Write the shortest codes that you can think of for the following tasks:

1. Given a string that represents a base X number, convert it to 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). (More details in Section 5.3.2).
2. Given a list of integers L of size up to $1M$ items, determine whether a value v exists in L by not using more than 20 comparisons? (More details in Section 2.2.1).
3. Given a date, determine what is the day (Monday, Tuesday, ..., Sunday) of that date? (e.g. 9 August 2010 – the launch date of the first edition of this book – is Monday).
4. Given a string, replace all 'special words' of length 3 with 3 stars "***". The 'special word' starts with a lowercase alphabet character and followed by two consecutive digits, e.g.
   S = "line: a70 and z72 will be replaced, but aa24 and a872 will not"
   will be transformed to
   S = "line: *** and *** will be replaced, but aa24 and a872 will not".
5. Write the shortest possible **Java code** to read in a double
   (e.g. `1.4732`, `15.324547327`, etc)
   and print it again, but now with minimum field width 7 and 3 digits after decimal point
   (e.g. `ss1.473` (where '`s`' denotes a space), `s15.325`, etc)
6. Generate all possible permutations of {0, 1, 2, ..., N-1}, for N = 10.
7. Generate all possible subsets of {0, 1, 2, ..., N-1}, for N = 20.

1.2.4 Tip 5: Master the Art of Testing Code

You thought that you have nailed a particular problem. You have identified its type, designed the algorithm for it, calculated the algorithm's time/space complexity - it will be within the time and memory limit given, and coded the algorithm. But, your solution is still not Accepted (AC).

Depending on the programming contest's type, you may or may not get credit by solving the problem partially. In ICPC, you will only get credit if your team's code solve **all** the judge's secret test cases, that's it, you get AC. Other responses like 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 IOI (2010-2011), the subtask scoring system is used. Test cases are grouped into subtasks, usually the simpler variants of the original task. You will only get the score of a subtask if your code solves all test cases that belong to that subtask. You are given *tokens* that you can use sparsely throughout the contest to see the judging result of your code.

In either case, you will need to be able to design good, educated, tricky test cases. The sample input-output given in problem description is by default too trivial and therefore not a good way for measuring the correctness of your code.

Rather than wasting submissions (and get time or point penalties) in ICPC or tokens in IOI by getting non AC responses, you may want to design some tricky test cases first, test it in your own machine, and ensure that your code is able to solve it correctly (otherwise, there is no point submitting your solution right?).

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

Here are some guidelines for designing good test cases, based on our experience:
These are the typical steps taken by the problem setters too.

1. Your test cases must include the sample input as you already have the answer given.
   Use `fc` in Windows or `diff` in UNIX to check your code's output against the sample output.
   Avoid manual comparison as we are not good in performing such task, especially for problems with delicate output format.

2. For multiple input test cases, you should include two identical sample test cases consecutively. Both must output the same known correct results.
   This is to check whether you have forgotten to initialize some variables, which will be easily identified if the 1st instance produces the correct output but the 2nd one does not.

3. Your test cases must include *large* cases.
   Increase the input size incrementally up to the maximum possible stated in problem description. Sometimes your program works for small input size, but behave wrongly (or slowly) when input size increases. Check for overflow, out of bounds, etc if that happens.

4. Your test cases must include the tricky corner cases.
   Think like the problem setter! Identify cases that are 'hidden' in the problem description. Some typical corner cases: N = 0, N = 1, N = maximum values allowed in problem description, N = negative values, etc. Think of the worst possible input for your algorithm.

5. Do not assume the input will always be nicely formatted if the problem description does not say so (especially for a badly written problem). Try inserting white spaces (spaces, tabs) in your input, and check whether your code is able to read in the values correctly (or crash).

6. Finally, generate large random test cases to see if your code terminates on time and still give reasonably ok output (the correctness is hard to verify here – this test is only to verify that your code runs within the time limit).

However, after all these careful steps, you may still get non-AC responses. In ICPC[6], you and your team can actually use the judge's response to determine your next action. With more experience in such contests, you will be able to make better judgment. See the next exercises:

Exercise 1.2.4: Situation judging (Mostly in ICPC setting. This is not so relevant in IOI).

1. You receive a WA response for a very easy problem. What should you do?
   (a) Abandon this problem and do another.
   (b) Improve the performance of your solution (optimize the code or use better algorithm).
   (c) Create tricky test cases and find the bug.
   (d) (In team contest): Ask another coder in your team to re-do this problem.

2. You receive a TLE response for an your $O(N^3)$ solution. However, maximum N is just 100. What should you do?
   (a) Abandon this problem and do another.
   (b) Improve the performance of your solution (optimize the code or use better algorithm).
   (c) Create tricky test cases and find the bug.

3. Follow up question (see question 2 above): What if maximum N is 100.000?

4. You receive an RTE response. Your code runs OK in your machine. What should you do?

5. 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 (your team) do?
   (a) Abandon the problem with WA code, switch to that other problem in 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 code.
   (c) (In ICPC): Print the WA code. Ask two other team members to scrutinize the code while you switch to that other problem in attempt to solve *two* more problems.

[6]In IOI 2010-2011, contestants have limited tokens that they can use sparingly to check the correctness of their submitted code. The exercise in this section is more towards ICPC style contest.

1.2.5 Tip 6: Practice and More Practice

Competitive programmers, like real athletes, must train themselves regularly and keep themselves 'programming-fit'. Thus in our last tip, we give a list of some websites that can help to improve your problem solving skill. Success comes as a result of a continuous journey to better yourself.

University of Valladolid (from Spain) Online Judge [28] contains past years ACM contest problems (usually local or regional) plus problems from other sources, including their own contest problems. You can solve these problems and submit your solutions to this Online Judge. The correctness of your program will be reported as soon as possible. Try solving the problems mentioned in this book and see your name on the top-500 authors rank list someday :-). At the point of writing (1 August 2011), Steven is ranked 64 (for solving 1180 problems) while Felix is ranked 54 (for solving 1239 problems) from ≈ 115266 UVa users and ≈ 2950 problems.

UVa's 'sister' online judge is the ACM ICPC Live Archive [20] that contains *almost all* recent ACM ICPC Regionals and World Finals problem sets since year 2000. Train here if you want to do well in future ICPCs.

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

USA Computing Olympiad has a very useful training website [29] and online contests for you to learn programming and problem solving. This one is geared more towards IOI participants. Go straight to their website, register your account, and train yourself.

Sphere Online Judge [35] is another online judge where qualified users can add problems too. This online judge is quite popular in countries like Poland, Brazil, and Vietnam. We use 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) [19] that consists of a few problems that should be solved in 1-2 hours. Afterwards, 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 in problem solving with higher rating as opposed to more diligent contestants who happen to solve 'more' easier problems.

1.2.6 Tip 7: Team Work (ICPC Only)

This last tip is not something that is teachable. But here are some ideas that may be worth trying:

- Practice coding on a blank paper (useful when your teammate is using the computer).
- Submit and print strategy. If your code is AC, ignore your printout. If it is still not AC, debug using that printout (and let your teammate use the computer for other problem).
- If your teammate is currently coding his algorithm, prepare challenges for his code by preparing hard corner test cases (and hopefully his code passes all those).
- The X-factor: Befriend your teammates *outside* the training sessions & actual competition.

1.3 Getting Started: The Ad Hoc Problems

We end this chapter by asking you to start with the first problem type in ICPCs and IOIs: the Ad Hoc problems. According to USACO [29], Ad Hoc problems are problems that 'cannot be classified anywhere else', where each problem description and its corresponding solution are 'unique'.

Ad Hoc problems almost always appear in a programming contest. Using a benchmark of total 10 problems, there may be 1-2 Ad Hoc problems in an ICPC. If the Ad Hoc problem is easy, it will usually be the first problem solved by the teams in a programming contest. But there exists Ad Hoc problems that are complicated to code and some teams will strategically defer solving them until the last hour. Assuming a 60-teams contest, your team is probably in the lower half (rank 30-60) if your team can *only* do this type of problem during an ICPC regional contest.

In IOI 2009 and 2010, there exists 1 easy task per competition day[7], which is usually an Ad Hoc task. If you are an IOI contestant, you will definitely not going to get any medal by only solving these 2 easy Ad Hoc tasks over 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.

To help you pick which problems to start with among the ≈ 2950 problems in UVa online judge [28] (and some other online judges), we have listed **many** Ad Hoc problems that we have solved into several sub-categories below. Each category still contains a lot of problems, so we *highlight* up to maximum three (3) **must try *** problems in each category. These are the problems that we think are more interesting or have better quality.

We believe that you can solve most of these problems *without* using advanced data structures or algorithms that will be discussed in the latter chapters. Many of these Ad Hoc problems are 'simple' but some of them maybe 'tricky'. Now, try to solve few problems from each category before reading the next chapter.

The categories:

- (Super) **Easy**
  You should get these problems AC[8] in under 7 minutes each!
  If you are new with competitive programming, we strongly recommend that you start your journey by solving some problems from this category.
- **Game (Card)**
  There are lots of Ad Hoc problems involving popular games. The first game type is related to cards. Usually you will need to parse the string input as normal cards have suits (D/Diamond/♢, C/Club/♣, H/Heart/♡, and S/Spades/♠) on top of the ranks (usually: 2 $<$ 3 $<$ ... $<$ 9 $<$ T/Ten $<$ J/Jack $<$ Q/Queen $<$ K/King $<$ A/Ace[9]). It may be a good idea to map these complicated 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, we work with the integer indices instead.
- **Game (Chess)**
  Another popular games that sometimes appear in programming contest problems are chess problems. Some of them are Ad Hoc (listed in this section). Some of them are combinatorial, like counting how many ways to put 8-queens in 8×8 chess board (listed in Chapter 3).
- **Game (Others)**
  Other than card and chess games, there are many other popular problems related to other games that make their way into programming contest problems: Tic Tac Toe, Rock-Paper-Scissors, Snakes/Ladders, BINGO, Bowling, and several others. Knowing the details of the game is helpful, but most of the game rules are given in the problem description to avoid disadvantaging contestants who have not played those games before.

[7] This is no longer true in IOI 2011.

[8] However, do not feel bad if you fail to do so. There are reasons why a code does not get AC response. Once you are more familiar with competitive programming, you will find that these problems are indeed super easy.

[9] In some other arrangement, A/Ace $<$ 2.

- The **'Josephus'**-type problems
  The Josephus problem is a classic problem where there are n people numbered from 1, 2, ..., n, standing in a circle. Every m-th person is going to be executed. Only the last remaining person will be saved (history said it was the person named Josephus). The smaller version of this problem can be solved with plain brute force. The larger ones require better solutions.

- Problems related to **Palindrome** or **Anagram**
  These are also classic problems.
  Palindrome is a word (or actually a sequence) that can be read the same way in either direction. The common strategy to check if a word is palindrome is to loop from the first character to the *middle* one and check if the first match the last, the second match the second last, and so on. Example: '**A**BCDCB**A**' is a palindrome.
  Anagram is a rearrangement of letters of a word (or phrase) to get another word (or phrase) using all the original letters. The common strategy to check if two words are anagram is to sort the letters of the words and compare the sorted letters. Example: wordA = 'cab', wordB = 'bca'. After sorting, wordA = 'abc' and wordB = 'abc' too, so they are anagram.

- Interesting **Real Life** Problems
  This is one of the most interesting category of problems in 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 problems is an extra motivation boost. Who knows you may also learn some new interesting knowledge from the problem description!

- Ad Hoc problems involving **Time**
  Date, time, calendar, ... All these are also real life problems. As said earlier, people usually get extra motivation when dealing with real life problems. Some of these problems will be much easier to solve if you have mastered the Java GregorianCalendar class as it has lots of library functions to deal with time.

- Just Ad Hoc
  Even after our efforts to sub-categorize the Ad Hoc problems, there are still many others that are too Ad Hoc to be given a specific sub-category. The problems listed in this sub-category are such problems. The solution for most problems is to simply follow/simulate the problem description carefully.

- Ad Hoc problems in **other chapters**
  There are many other Ad Hoc problems which we spread to other chapters, especially because they require some more knowledge on top of basic programming skills.

  - Ad Hoc problems involving the usage of basic linear data structures, especially arrays are listed in Section 2.2.1.
  - Ad Hoc problems involving mathematical computations are listed in Section 5.2.
  - Ad Hoc problems involving processing of strings are listed in Section 6.3.
  - Ad Hoc problems involving basic geometry skills are listed in Section 7.2.

Tips: After solving some number of programming problems, you will encounter some pattern. From a C/C++ perspective, those pattern are: 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 all those in a header file 'competitive.h'. Now, every time he wants to solve another problem, he just need to open a new *.c or *.cpp file, and type `#include<competitive.h>`.

Programming Exercises related to Ad Hoc problems:

- Easy Problems in UVa Online Judge (under 7 minutes)
  1. UVa 00272 - TEX Quotes (simply replace all double quotes to TEX() style quotes)
  2. UVa 00494 - Kindergarten Counting Game (count frequency of words in a line[10])
  3. UVa 00499 - What's The Frequency ... (ad hoc, 1D array manipulation)
  4. UVa 10300 - Ecological Premium (straightforward, ignore 'total animal')
  5. **UVa 10420 - List of Conquests *** (simple frequency counting, use `map`)
  6. UVa 10550 - Combination Lock (simple, do as asked)
  7. UVa 11044 - Searching for Nessy (one liner code/formula exists)
  8. UVa 11172 - Relational Operators (ad hoc, very easy, one liner)
  9. UVa 11332 - Summing Digits (simple recursions, $O(1)$ solution exists (modulo 9))
  10. UVa 11498 - Division of Nlogonia (straightforward problem; use one if-else statement)
  11. UVa 11547 - Automatic Answer (one liner $O(1)$ solution exists)
  12. **UVa 11559 - Event Planning *** (answer can be found by reading input data once)
  13. UVa 11727 - Cost Cutting (sort the 3 numbers and get the median)
  14. UVa 11764 - Jumping Mario (one linear scan to count high and low jumps)
  15. **UVa 11799 - Horror Dash *** (one linear scan to find the max value)
  16. UVa 11942 - Lumberjack Sequencing (check if input is sorted ascending/descending)
  17. UVa 12015 - Google is Feeling Lucky (go through the list twice: get max, print max)
- 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)
  4. **UVa 10205 - Stack 'em Up *** (card game)
  5. **UVa 10646 - What is the Card? *** (shuffle cards with some rule; get certain card)
  6. UVa 11225 - Tarot scores (another card game)
  7. UVa 11678 - Card's Exchange (actually just array manipulation problem)
- 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)
  6. UVa 10849 - Move the bishop (chess)
  7. UVa 11494 - Queen (ad hoc, chess)
- Game (Others)
  1. UVa 00220 - Othello (follow the game rules, a bit tedious)
  2. UVa 00227 - Puzzle (parse the input, array manipulation)
  3. UVa 00232 - Crossword Answers (another complex array manipulation problem)
  4. UVa 00339 - SameGame Simulation (follow problem description)
  5. UVa 00340 - Master-Mind Hints (determine strong and weak matches)
  6. UVa 00489 - Hangman Judge (just do as asked)
  7. **UVa 00584 - Bowling *** (simulation, games, test your reading comprehension)
  8. UVa 00647 - Chutes and Ladders (child board game, similar to UVa 11459)
  9. **UVa 10189 - Minesweeper *** (simulate Minesweeper game, similar to UVa 10279)
  10. UVa 10279 - Mine Sweeper (2D array helps, similar to UVa 10189)
  11. UVa 10363 - Tic Tac Toe (check validity of Tic Tac Toe game, tricky)

[10]Delimiter are the non-alphabetical characters. Can be solved 'elegantly' with Java Regular Expression.

12. UVa 10409 - Die Game (just simulate the die movement)
13. UVa 10443 - Rock, Scissors, Paper (2D arrays manipulation)
14. UVa 10530 - Guessing Game (use 1D flag array)
15. UVa 10813 - Traditional BINGO (follow the problem description)
16. UVa 10903 - Rock-Paper-Scissors ... (count win+losses, output win average)
17. **UVa 11459 - Snakes and Ladders *** (simulate the process, similar to UVa 647)

- Josephus
  1. UVa 00130 - Roman Roulette (brute force, similar to UVa 133)
  2. UVa 00133 - The Dole Queue (brute force, similar to UVa 130)
  3. **UVa 00151 - Power Crisis *** (brute force, similar to UVa 440)
  4. **UVa 00305 - Joseph *** (the answer can be precalculated)
  5. UVa 00402 - M*A*S*H (simulation problem)
  6. UVa 00440 - Eeny Meeny Moo (brute force, similar to UVa 151)
  7. **UVa 10015 - Joseph's Cousin *** (modified Josephus problem, variant of UVa 305)
- Palindrome / 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 00353 - Pesky Palindromes (brute force all substring)
  5. UVa 00401 - Palindromes (simple palindrome check)
  6. UVa 00454 - Anagrams (anagram)
  7. UVa 00630 - Anagrams (II) (ad hoc, string)
  8. UVa 10018 - Reverse and Add (ad hoc, math, palindrome check)
  9. UVa 10098 - Generating Fast, Sorted ... (very similar to UVa 195)
  10. UVa 10945 - Mother Bear (palindrome)
  11. **UVa 11221 - Magic Square Palindrome *** (not just a word, but a matrix)
  12. UVa 11309 - Counting Chaos (palindrome check)
- Interesting Real Life Problems
  1. **UVa 00161 - Traffic Lights *** (you see this problem every time you hit the road)
  2. UVa 00346 - Getting Chorded (musical chord, major/minor)
  3. UVa 00400 - Unix ls (this command is very popular among UNIX users)
  4. UVa 00403 - Postscript (emulation of printer driver)
  5. UVa 00448 - OOPS (tedious 'hexadecimal' to 'assembly language' conversion)
  6. UVa 00538 - Balancing Bank Accounts (the story in the problem really happens)
  7. **UVa 00637 - Booklet Printing *** (application in printer driver software)
  8. UVa 00706 - LC-Display (this is what we typically see in old computer monitors)
  9. UVa 10082 - WERTYU (this keyboard typing error happens to us sometimes)
  10. UVa 10191 - Longest Nap (you may want to apply this for your own schedule)
  11. UVa 10415 - Eb Alto Saxophone Player (a problem about musical instrument)
  12. UVa 10528 - Major Scales (the music knowledge is given in the problem description)
  13. UVa 10554 - Calories from Fat (for those who are concerned with their weights)
  14. UVa 10659 - Fitting Text into Slides (typical presentation programs do this)
  15. **UVa 10812 - Beat the Spread *** (be careful with boundary cases!)
  16. UVa 11223 - O: dah, dah, dah (tedious morse code conversion problem)
  17. UVa 11530 - SMS Typing (many handphone users encounter this problem everyday)
  18. UVa 11743 - Credit Check (Luhn's algorithm to check credit card number)
  19. UVa 11984 - A Change in Thermal Unit (F° to C° conversion and vice versa)
- 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 00837 - Y3K *** (use Java GregorianCalendar; similar to UVa 11356)
5. UVa 10070 - Leap Year or Not Leap Year ... (more than just ordinary leap years)
6. UVa 10371 - Time Zones (follow the problem description)
7. UVa 10683 - The decadary watch (simple clock system conversion)
8. UVa 11219 - How old are you? (be careful with boundary cases!)
9. UVa 11356 - Dates (very easy if you use Java GregorianCalendar)
10. UVa 11650 - Mirror Clock (some small mathematics required)
11. UVa 11677 - Alarm Clock (similar idea with UVa 11650)
12. **UVa 11947 - Cancer or Scorpio *** (much easier with Java GregorianCalendar)
13. UVa 11958 - Coming Home (be careful with 'past midnight')
14. UVa 12019 - Doom's Day Algorithm (Java Gregorian Calendar; get DAY_OF_WEEK)

- Just Ad Hoc

1. UVa 00101 - The Blocks Problem (simulation)
2. UVa 00114 - Simulation Wizardry (simulation)
3. UVa 00119 - Greedy Gift Givers (simulate give and receive process)
4. UVa 00121 - Pipe Fitters (ad hoc)
5. UVa 00139 - Telephone Tangles (adhoc string, simulation)
6. UVa 00141 - The Spot Game (simulation)
7. UVa 00144 - Student Grants (simulation)
8. UVa 00145 - Gondwanaland Telecom (simply calculate the charges)
9. UVa 00187 - Transaction Processing (simulation)
10. UVa 00335 - Processing MX Records (simulation)
11. UVa 00337 - Interpreting Control Sequences (simulation, output related)
12. UVa 00349 - Transferable Voting (II) (simulation)
13. UVa 00362 - 18,000 Seconds Remaining (simulation)
14. UVa 00379 - HI-Q (simulation)
15. UVa 00381 - Making the Grade (simulation)
16. UVa 00405 - Message Routing (simulation)
17. UVa 00434 - Matty's Blocks (ad hoc)
18. UVa 00457 - Linear Cellular Automata (simulation)
19. UVa 00496 - Simply Subsets (ad hoc, set manipulation)
20. **UVa 00556 - Amazing *** (simulation)
21. UVa 00573 - The Snail (simulation, beware of boundary cases!)
22. UVa 00608 - Counterfeit Dollar (ad hoc)
23. UVa 00621 - Secret Research (case analysis for 4 possible outputs)
24. UVa 00661 - Blowing Fuses (simulation)
25. **UVa 00978 - Lemmings Battle *** (simulation, use `multiset`)
26. UVa 10019 - Funny Encryption Method (not hard, find the pattern)
27. UVa 10033 - Interpreter (adhoc, simulation)
28. **UVa 10114 - Loansome Car Buyer *** (simulate loan payment+car depreciation)
29. UVa 10141 - Request for Proposal (this problem can be solved with one linear scan)
30. UVa 10142 - Australian Voting (simulation)
31. UVa 10188 - Automated Judge Script (simulation)
32. UVa 10267 - Graphical Editor (simulation)
33. UVa 10324 - Zeros and Ones (simplified using 1D array: change counter)
34. UVa 10424 - Love Calculator (just do as asked)
35. UVa 10707 - 2D - Nim (check graph isomorphism, a bit hard)
36. UVa 10865 - Brownie Points (simulation)
37. UVa 10919 - Prerequisites? (process the requirements as the input is read)
38. UVa 10963 - The Swallowing Ground (not hard)
39. UVa 11140 - Little Ali's Little Brother (ad hoc)
40. UVa 11507 - Bender B. Rodriguez Problem (simulation)

41. UVa 11586 - Train Tracks (TLE if brute force, find the pattern)
42. UVa 11661 - Burger Time? (linear scan)
43. UVa 11679 - Sub-prime (check if after simulation all banks still have ≥ 0 reserve)
44. UVa 11687 - Digits (simulation, straightforward)
45. UVa 11717 - Energy Saving Microcontroller (tricky simulation)
46. UVa 11850 - Alaska (ad hoc)
47. UVa 11917 - Do Your Own Homework (ad hoc)
48. UVa 11946 - Code Number (ad hoc)
49. UVa 11956 - Brain**** (simulation; ignore '.')
50. IOI 2009 - Garage
51. IOI 2009 - POI
52. IOI 2010 - Cluedo (use 3 pointers)
53. IOI 2010 - Memory (use 2 linear pass)
54. LA 2189 - Mobile Casanova (Dhaka06)
55. LA 3012 - All Integer Average (Dhaka04)
56. LA 3996 - Digit Counting (Danang07)
57. LA 4147 - Jollybee Tournament (Jakarta08)
58. LA 4202 - Schedule of a Married Man (Dhaka08)
59. LA 4786 - Barcodes (World Finals Harbin10)
60. LA 4995 - Language Detection (KualaLumpur10)

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

1.4 Chapter Notes

This and subsequent chapters are supported by many text books (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 lots of typing games that are available online.
- Tip 2 is an adaptation from the introduction text in USACO training gateway [29].
- More details about Tip 3 can be found in many CS books, e.g. Chapter 1-5, 17 of [3].
- Online references for Tip 4 are:
  `http://www.cppreference.com` and `http://www.sgi.com/tech/stl/` for C++ STL;
  `http://java.sun.com/javase/6/docs/api` for Java API.
- For more insights to do better testing (Tip 5),
  a little detour to software engineering books may be worth trying.
- There are many other Online Judges apart from those mentioned in Tip 6, e.g.
  - POJ `http://acm.pku.edu.cn/JudgeOnline`,
  - TOJ `http://acm.tju.edu.cn/toj`,
  - ZOJ `http://acm.zju.edu.cn/onlinejudge/`,
  - Ural/Timus OJ `http://acm.timus.ru`, etc.
- For a note regarding team contest (Tip 7), read [7].

In this chapter, we have introduced the world of competitive programming to you. However, you cannot say that you are a competitive programmer if you can only solve Ad Hoc problems in every programming contest. Therefore, we do hope that you enjoy the ride and continue reading and learning *the other chapters* of this book, enthusiastically. Once you have finished reading this book, re-read it one more time. On the second round, attempt the various written exercises and the ≈ 1198 programming exercises as many as possible.

There are ≈ **149 UVa (+ 11 others) programming exercises** discussed in this chapter.
(Only 34 in the first edition, a 371% increase).

There are **19 pages** in this chapter.
(Only 13 in the first edition, a 46% increase).


ACM-ICPC Asia Phuket Regional
3-4 November 2009 Phuket
University Phuket Campus
L-R: Adhiraj, Steven
ACM-ICPC Asia Phuket Regional
Programming Contest 2009
Prince of Songkla University Phuket Campus
30,000.-
L-R: Victor, Hung, Dr CJ Hwang, Phong, Duc, Tien


THAILAND
2011
L-R: Hubert, Chuanqi, Zi Chun, Steven, Zhan Xiong, Raymond, Mr Cheong

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

Data structure is 'a way to store and organize data' in order to support efficient insertions, searches, deletions, queries, and/or updates. Although a data structure by itself does not solve the given programming problem (the algorithm operating on it does), using the most efficient data structure for the given 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 on a different context, as we will see in the discussion below. Familiarity with the data structures and libraries discussed in this chapter is a must in order to understand 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, and thus we do **not** review them again in this book. In this book, we highlight the fact that they all have built-in libraries in C++ STL and Java API[1]. Therefore, if you feel that you are not sure with any of the terms or data structures mentioned in Section 2.2, please pause reading this book, quickly explore and learn that particular term in the reference books, e.g. [2][2], and only resume when you get the *basic ideas* of those data structures.

Note that for competitive programming, you just have to be able to *use* the correct data structure to solve the given contest problem. You just have to know the strengths, weaknesses, and typical time/space complexities of the data structure. Its theoretical background is good to know, but can be skipped. This is *not* a good practice but it works. For example, many (younger) contestants may have used the efficient C++ STL `map` (in Java: use `TreeMap`) to store dynamic collection of key-data pairs without an understanding that the underlying data structure is a *balanced Binary Search Tree* – which is typically taught in year 1 Computer Science curriculum.

This chapter is divided into two parts. Section 2.2 contains basic data structures with their basic operations that already have built-in libraries. The discussion of each data structure in this section is very brief. Section 2.3 contains *more* data structures for which currently we have to build our own libraries. Because of this, Section 2.3 has more detailed discussions than Section 2.2.

The best way to learn the material in this chapter is by examining the example codes. We have written lots of programming examples in both C++ and Java. The source codes are available in `https://sites.google.com/site/stevenhalim/home/material`. The reference to each example code is indicated the body text as a box like below.

Download codes from `https://sites.google.com/site/stevenhalim/home/material`

[1]In the second edition of this book, we *still* write most example codes from C++ perspective. However, the Java version of the C++ codes can be found in the supporting website of this book.

[2]Materials in Section 2.2 are usually taught in level-1 'data structures and algorithms' course in CS curriculum. High school students who are aspiring to take part in the IOI are encouraged to do self-study on these material.

2.2 Data Structures with Built-in Libraries

2.2.1 Linear Data Structures

A data structure is classified as *linear* if its elements form a sequence. Mastery of all these basic linear data structures below is a must to do well 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*. As the maximum input size is typically mentioned in a programming problem, the declared array size will usually be the value of the given constraint plus a small extra buffer for safety. Typical array dimensions are: 1D, 2D, and rarely goes beyond 3D. Typical array operations are: accessing certain indices, sorting, linearly scanning, or binary searching the sorted array.
- Resizeable Array a.k.a. Vector: C++ STL `vector` (Java `ArrayList` (faster) or `Vector`)
  Same as static array but has auto-resize feature. It is better to use `vector` in place of array if array size is unknown before running the program. Usually, we initialize the size with some guess value for better performance. Typical operations are: `push_back()`, `at()`, `[]` operator, `assign()`, `clear()`, `erase()`, and using `iterator` to scan the content of the `vector`.

Example codes: `ch2_01_staticarray_stl_vector.cpp`; `ch2_01_staticarray_Vector.java`

Quick Detour on Efficient Sorting and Searching in Static/Resize-able Array

There are two central operations commonly performed on array: **sorting** and **searching**. It is noteworthy to mention that these two operations are well supported in C++ and Java.

There are *many* sorting algorithms mentioned in CS textbooks [3, 24, 32] which we classify as:

1. $O(n^2)$ comparison-based sorting algorithms: Bubble/Selection/Insertion Sort.
   These algorithms are slow and usually avoided, but understanding them is important.
2. $O(n \log n)$ comparison-based sorting algorithms: Merge/Heap/Random Quick Sort.
   We can use `sort`, `partial_sort`, or `stable_sort` in C++ STL `algorithm` (Java `Collections.sort`) to achieve this purpose. 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.
   These special purpose algorithms are good to know, as they can speed up the sorting time if the problem has special characteristics, like small range of integers for Counting Sort, but they rarely appear in programming contests.

There are basically three ways to search for an item in Array which we classify as:

1. $O(n)$ Linear Search from index 0 to index $n-1$ (avoid this in programming contests).
2. $O(\log n)$ Binary Search: use `lower_bound` in C++ STL algorithm (Java `Collections.binarySearch`). If the input is unsorted, it is fruitful to sort it just once using an $O(n \log n)$ sorting algorithm above in order to use Binary Search *many times*.
3. $O(1)$ with Hashing (but we can live without hashing for most contest problems).

Example codes: `ch2_02_stl_algorithm.cpp`; `ch2_02_Collections.java`

- Linked List: C++ STL `list` (Java `LinkedList`)
  Although this data structure almost always appears in data structure & algorithm textbooks, Linked List is usually avoided in typical contest problems. Reasons: It involves pointers and slow for accessing data as it has to be performed from the head or tail of a list. In this book, almost all form of Linked List is replaced by the more flexible C++ STL `vector` (Java `Vector`). The only exception is probably UVa 11988 - Broken Keyboard (a.k.a. Beiju Text).

- Stack: C++ STL `stack` (Java `Stack`)
  This data structure is used as a part of algorithm to solve a certain problem (e.g. Postfix calculation, bracket matching, Graham's scan in Section 7.3.7). Stack only allows insertion (push) and deletion (pop) from the top only. This behavior is called Last In First Out (LIFO) as with normal stacks in the real world. Typical operations are `push()`/`pop()` (insert/remove from top of stack), `top()` (obtain content from the top of stack), `empty()`.

- Queue: C++ STL `queue` (Java `Queue`[3])
  This data structure is used in algorithms like Breadth First Search (BFS) (Section 4.2.2). A queue only allows insertion (enqueue) from back (tail), and only allows deletion (dequeue) from front (head). This behavior is called First In First Out (FIFO), similar to normal queues in the real world. Typical operations are `push()`/`pop()` (insert from back/remove from front of queue), `front()`/`back()` (obtain content from the front/back of queue), `empty()`.

Example codes: `ch2_03_stl_stack_queue.cpp`; `ch2_03_Stack_Queue.java`

- Lightweight Set of Boolean (a.k.a bitmask) (native support in both C/C++ and Java)
  An integer is stored in computer memory as a sequence of bits. This can be used to represent *lightweight* small set of Boolean. All operations on this set of Boolean uses bit manipulation of the corresponding integer, which makes it much faster than C++ STL `vector<bool>` or `bitset`. *Some* important operations that we used in this book are shown below.

  1. Representation: 32-bit (or 64-bit) integer, up to 32 (or 64) items. Without loss of generality, all our examples below assume that we use a 32-bit integer called S.

     ```
     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)
     ```

  2. To multiply/divide an integer by 2, we just need to shift left/right the corresponding bits of the integer, respectively. This operation (especially shift left) is important for the next few operations below.

     ```
     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) <- last '1' is gone
     ```

  3. To set/turn on the `j`-th item of the set, use `S |= (1 << j)`.

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

  4. To check if the `j`-th item of the set is on (or off), use `T = S & (1 << j)`. If `T = 0`, then the `j`-th item of the set is off. If `T != 0`, then the `j`-th item of the set is on.

     ```
     S = 42 (base 10) = 101010 (base 2)
     j = 3, 1 << j    = 001000 <- bit '1' is shifted to left 3 times
                        -------- AND (only true if both bits are true)
     T = 8 (base 10)  = 001000 (base 2) -> not zero, so the 3rd item is on
     S = 42 (base 10) = 101010 (base 2)
     j = 2, 1 << j    = 000100 <- bit '1' is shifted to left 2 times
                        -------- AND
     T = 0 (base 10)  = 000000 (base 2) -> zero, so the 2rd item is off
     ```

[3]Java Queue is only an *interface* that must be instantiated with Java LinkedList. See our sample codes.

5. To clear/turn off the `j`-th item of the set, use[4] `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) // this new value 40 is updated to S
```

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

```
S = 40 (base 10) = 101000 (base 2)
j = 2, (1 << j)  = 000100 <- bit '1' is shifted to left 4 times
                   -------- XOR (only true if both bits are different)
S = 44 (base 10) = 101100 (base 2) // this new value 44 is updated to S
```

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

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

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

```
Example for n = 6
S + 1 = 64 (base 10) = 1000000 <- bit '1' is shifted to left 6 times
                              1
                       --------- -
S     = 63 (base 10) =  111111 (base 2)
```

Many bit manipulation operations are written as macros in our C/C++ example codes (but written as usual in our Java example codes as Java does not support macro).

Example codes: `ch2_04_bit_manipulation.cpp`; `ch2_04_bit_manipulation.java`

Programming exercises to practice using linear data structures (and algorithms) with libraries:

- Basic Data Structures[5]
  1. UVa 00394 - Mapmaker (1D array manipulation)
  2. UVa 00414 - Machined Surfaces (1D array manipulation, get longest stretch of 'B's)
  3. UVa 00466 - Mirror Mirror (2D array manipulation)
  4. UVa 00467 - Synching Signals (linear scan on 1D array)
  5. UVa 00482 - Permutation Arrays (1D array manipulation)
  6. UVa 00541 - Error Correction (2d array manipulation)
  7. UVa 00591 - Box of Bricks (1D array manipulation)
  8. UVa 00594 - One Little, Two Little ... (manipulate bit string easily with `bitset`)
  9. UVa 00700 - Date Bugs (can be solved with `bitset`)
  10. UVa 10016 - Flip-flop the Squarelotron (tedious 2D array manipulation)
  11. UVa 10038 - Jolly Jumpers (1D array manipulation)
  12. UVa 10050 - Hartals (1D boolean array manipulation)
  13. UVa 10260 - Soundex (Direct Addressing Table for soundex code mapping)
  14. UVa 10703 - Free spots (just use small 2D array of size 500×500)
  15. UVa 10855 - Rotated squares (2D string array manipulation, 90^o clockwise rotation)
  16. UVa 10920 - Spiral Tap (simulate the process)

[4] Use brackets a lot when doing bit manipulation to avoid unnecessary bugs with operator precedence.

[5] Various data structures under this subcategory are: Static array, C++ STL `vector`, `bitset`, lightweight set of Boolean (bit manipulation of integers), Direct Addressing Table, `list` (Java `Vector`, `ArrayList`, `LinkedList`).

17. UVa 10978 - Let's Play Magic (1D string array manipulation)
18. UVa 11040 - Add bricks in the wall (non trivial 2D array manipulation)
19. UVa 11192 - Group Reverse (1D character array manipulation)
20. UVa 11222 - Only I did it (use several 1D arrays to simplify this problem)
21. **UVa 11340 - Newspaper *** (Direct Addressing Table)
22. UVa 11349 - Symmetric Matrix (2D array manipulation)
23. UVa 11360 - Have Fun with Matrices (2D array manipulation)
24. UVa 11364 - Parking (1D array manipulation)
25. UVa 11496 - Musical Loop (1D array manipulation)
26. **UVa 11581 - Grid Successors *** (2D array manipulation)
27. UVa 11608 - No Problem (1D array manipulation)
28. UVa 11760 - Brother Arif, Please ... (row+col checks are separate; use two 1D arrays)
29. UVa 11835 - Formula 1 (2D array manipulation)
30. UVa 11933 - Splitting Numbers (an exercise for bit manipulation)
31. **UVa 11988 - Broken Keyboard ... *** (rare problem using linked list)
32. IOI 2011 - Pigeons (this problem is simpler if contestants know bit manipulation)

- 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 10194 - Football a.k.a. Soccer (multi-fields sorting, use `sort`)
  4. **UVa 10258 - Contest Scoreboard *** (multi-fields sorting, use `sort`)
  5. UVa 10880 - Colin and Ryan (use `sort`)
  6. UVa 10905 - Children's Game (modified comparison function, use `sort`)
  7. UVa 11039 - Building Designing (use `sort` then count different signs)
  8. UVa 11321 - Sort Sort and Sort (be careful with negative mod!)
  9. **UVa 11588 - Image Coding *** (`sort` simplifies the problem)
  10. UVa 11621 - Small Factors (generate numbers with factor 2 and/or 3, `sort`, `upper_bound`)
  11. UVa 11777 - Automate the Grades (`sort` simplifies the problem)
  12. UVa 11824 - A Minimum Land Price (`sort` simplifies the problem)
  13. LA 3173 - Wordfish (Manila06) (STL `next_permutation`, `prev_permutation`)

- Sorting-related problems
  1. UVa 00110 - Meta-loopless sort (Ad Hoc)
  2. UVa 00299 - Train Swapping (inversion index[6] problem; $O(n^2)$ bubble sort)
  3. UVa 00450 - Little Black Book (tedious sorting problem)
  4. UVa 00612 - DNA Sorting (inversion index + `stable_sort`)
  5. **UVa 00855 - Lunch in Grid City *** (sort, median)
  6. UVa 10107 - What is the Median? (sort, median; or `nth_element` in <`algorithm`>)
  7. UVa 10327 - Flip Sort (inversion index - solvable with $O(n^2)$ bubble sort)
  8. UVa 10810 - Ultra Quicksort (inversion index - requires $O(n \log n)$ merge sort[7])
  9. **UVa 11462 - Age Sort *** (counting sort problem[8], discussed in [3].)
  10. UVa 11495 - Bubbles and Buckets (inversion index - requires $O(n \log n)$ merge sort)
  11. UVa 11714 - Blind Sorting (decision tree to find min and 2nd min)
  12. **UVa 11858 - Frosh Week *** (inversion index; $O(n \log n)$ merge sort; long long)

[6]Count how many 'bubble sort' swaps (swap between pair of consecutive items) are needed to make the list sorted. For example, if the array content is {4, 1, 2, 3}, we need 3 'bubble sort' swaps to make this array sorted.

[7]The $O(n \log n)$ solution for this inversion index problem is to modify merge sort. Let assume we sort ascending. During the merge process, if the front of the right sorted subarray is taken first rather than the front of the left sorted subarray, we say that 'inversion occurs'. Add inversion index by the size of the current left subarray.

[8]The idea of counting sort is simple, but it is only applicable to sort an array of integers with small range (like 'human age' of `[1..99]` years in UVa 11462). Do one pass through the array to count the frequency of each integer. Suppose the array is {`0, 1, 0, 0, 3, 1`}. With one pass, we know that `freq[0] = 3, freq[1] = 2, freq[2] = 0, freq[3] = 1`. Then we simply display 0 three times, 1 two times, 2 zero time (no output), 3 one time, to get the sorted array {`0, 0, 0, 1, 1, 3`}. Counting sort is used in our Suffix Array implementation in Section 6.6.4.

- C++ STL `stack` (Java `Stack`)
  1. UVa 00120 - Stacks Of Flapjacks (Ad Hoc, pancake sorting)
  2. UVa 00127 - "Accordian" Patience (shuffling `stack`)
  3. **UVa 00514 - Rails *** (use `stack` to simulate the process)
  4. UVa 00551 - Nesting a Bunch of Brackets (bracket matching with `stack`, classic)
  5. UVa 00673 - Parentheses Balance (similar to UVa 551, classic)
  6. **UVa 00727 - Equation *** (infix to postfix conversion, classic)
  7. UVa 00732 - Anagram by Stack (use `stack` to simulate the process)
  8. UVa 10858 - Unique Factorization (use `stack` to help solving this problem)
  9. **UVa 11111 - Generalized Matrioshkas *** (bracket matching with some twists)

  Also see: `stack` inside any recursive function calls (implicitly)
- C++ STL `queue` (Java `Queue`)
  1. UVa 00540 - Team Queue (modified 'queue')
  2. **UVa 10172 - The Lonesome Cargo ... *** (simulation with 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. IOI 2011 - Hottest (practice task; 'sliding window'; double ended queue/`deque`)
  7. IOI 2011 - Ricehub (solvable with 'sliding window'; double ended queue/`deque`)

  Also see: `queue` inside BFS (see Section 4.2.2)

2.2.2 Non-Linear Data Structures

For some problems, there are better ways to organize the data other than storing the data linearly. With efficient implementation of non-linear data structures shown below, you can operate on the data in a faster manner, which can speed up the algorithms that use them.

For example, if you want to store a dynamic collection of pairs (e.g. name → index pairs), then using C++ STL `map` below can give you $O(\log n)$ performance for insertion/search/deletion with just few lines of codes whereas storing the same information inside one static array of `struct` may require $O(n)$ insertion/search/deletion and you have to code it by yourself.

- Balanced Binary Search Tree (BST): C++ STL `map/set` (Java `TreeMap/TreeSet`)
  BST is one way to organize data as a tree structure. In each subtree rooted at x, this 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 a spirit of Divide and Conquer (also see Section 3.3). Organizing the data like this (see Figure 2.1, left) allows $O(\log n)$ insertion, search, and deletion as only $O(\log n)$ worst case root-to-leaf scan is needed to perform these operations (see [3, 2] for details) – but this only works if the BST is balanced.

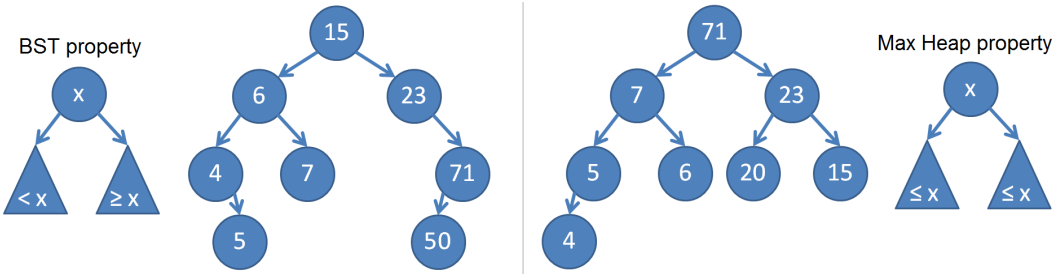


Figure 2.1: Examples of BST (Left) and (Max) Heap (Right)

Implementing a *bug-free* balanced BST like the Adelson-Velskii Landis (AVL)[9] or Red-Black

[9]AVL tree is the first self-balancing BST to be invented. AVL tree has one more property on top of BST property: The heights of the two subtrees of any vertex can only differ *by at most one*. Rebalancing act is performed (when necessary) during insertions and deletions to maintain this property.

(RB)[10] Tree is tedious and hard to do under time constrained contest environment (unless you have prepared a library code beforehand). Fortunately, C++ STL has `map` and `set` (Java `TreeMap` and `TreeSet`) which are *usually* the implementation of RB Tree which guarantee all insertion/search/deletion operations are done in $O(\log n)$. By mastering these two STLs, you can save a lot of precious coding time during contests[11]! The difference between these two data structures is simple: C++ STL `map` stores (key → data) pair whereas C++ STL `set` only stores the key.

Example codes: `ch2_05_stl_map_set.cpp`; `ch2_05_TreeMap_TreeSet.java`

- Heap: C++ STL `priority_queue` (Java `PriorityQueue`)
  Heap is another way to organize data as a tree structure. Heap is also a binary tree like BST but it must be *complete*[12]. Instead of enforcing the BST property, (Max) Heap enforces the Heap property: In each subtree rooted at x, items on the left **and** the right subtrees of x are smaller than (or equal to) x (see Figure 2.1, right). This is also a spirit of Divide and Conquer (also see Section 3.3). This property guarantees that the top of the heap is the maximum element. There is usually no notion of 'search' in Heap, but only deletion of the maximum element and insertion of new item – which can be easily done by traversing a $O(\log n)$ root-to-leaf or leaf-to-root path (see [3, 2] for details).

  (Max) Heap is useful to model a Priority Queue, where item with the highest priority (the maximum element) can be dequeued from the priority queue in $O(\log n)$ and new item can be enqueue into the priority queue also in $O(\log n)$. The implementation of `priority_queue` is available in C++ STL `queue` (Java `PriorityQueue`). Priority Queue is an important component in algorithms like Prim's (and Kruskal's) algorithm for Minimum Spanning Tree (MST) problem (Section 4.3) and Dijkstra's algorithm for Single-Source Shortest Paths (SSSP) problem (Section 4.4.3).

  This data structure is also used to perform `partial_sort` in C++ STL `algorithm`. This is done by dequeuing the maximum element k times where k is the number of the top most items to be sorted. As each dequeue operation is $O(\log n)$, `partial_sort` has $O(k \log n)$ time complexity[13]. When $k = n$, this algorithm is called heap sort. Note that although the time complexity of heap sort is $O(n \log n)$, heap sort is usually slower than quick sort because heap operations have to access data that are stored in very different indices, thus decreasing the performance of cache memory.

  Example codes: `ch2_06_stl_priority_queue.cpp`; `ch2_06_PriorityQueue.java`

- Hash Table: no native C++ STL support (Java `HashMap`/`HashSet`/`HashTable`)
  Hash Table is another non-linear data structures, but we do not recommend using it in programming contests unless necessary. Reasons: Designing a good performing hash function is quite tricky and there is no native C++ STL support for it[14]. Moreover C++ STL `map` or `set` (Java `TreeMap` or `TreeSet`) are usually good enough as the typical input size of programming contest problems are usually not more than $1M$, making the performance of $O(1)$ for Hash Table and $O(\log 1M)$ for balanced BST do not differ by much.

[10] Red-Black tree is another self-balancing BST. Every vertex has color attribute: either red or black. The root is black. All leaves are black. 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, RB tree will maintain all these properties to keep the tree balanced.

[11] However, for some tricky contest problems where you have to 'augment your data structure' (see Chapter 14 of [3]), AVL Tree knowledge is needed as C++ STL `map` or `set` cannot be augmented easily.

[12] Complete binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all vertices are as far left as possible. Complete binary tree can be stored in a compact array.

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

[14] Java has Hash related classes, but we decide not to use them *yet* in this book.

However, a form of Hash Table is actually used in contests, namely 'Direct Addressing Table' (DAT), where the key itself is the index, bypassing the need of a 'hash 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'. In this case, we do not need to use C++ STL `map` or any form of hashing as the key itself (the ASCII character) is unique and sufficient to determine the appropriate index in the array of size 256.

Exercise 2.2.2.1: Which non linear data structure that you will use if you have to support the following three dynamic operations: 1) many insertions, 2) many deletions, and 3) many requests to print data currently in the structure in sorted order.

Exercise 2.2.2.2: There are **M strings**. **N** of them are unique (obviously $\mathbf{N} \leq \mathbf{M}$). Which non linear data structure that you will use if you have to index these **M** strings with integers from `[0..N-1]`? The indexing criteria is up to you, but similar strings must be given similar index!

Programming exercises to practice using non-linear data structures with libraries:

- C++ STL `priority_queue` (Java `PriorityQueue`)[15]
  1. **UVa 10954 - Add All *** (use `priority_queue`, greedy)
  2. **UVa 11995 - I Can Guess ... *** (simulation using `stack, queue, priority_queue`)
  3. LA 3135 - Argus (Beijing04) (use `priority_queue`)

  Also see: `priority_queue` inside topological sort variant (see Section 4.2.1)

  Also see: `priority_queue` inside Kruskal's algorithm[16] (see Section 4.3.2)

  Also see: `priority_queue` inside Prim's algorithm (see Section 4.3.3)

  Also see: `priority_queue` inside Dijkstra's algorithm (see Section 4.4.3)
- C++ STL `map`/`set` (Java `TreeMap`/`TreeSet`)
  1. UVa 00417 - Word Index (generate all words, add to `map` for auto sorting)
  2. UVa 00484 - The Department of ... (`map`: maintain freq, `vector`: input order)
  3. UVa 00501 - Black Box (use `multiset` with efficient iterator manipulation)
  4. UVa 00642 - Word Amalgamation (use `map`)
  5. UVa 00755 - 487-3279 (use `map` to simplify this problem)
  6. UVa 00860 - Entropy Text Analyzer (use `map` for frequency counting)
  7. **UVa 10226 - Hardwood Species *** (use `map`; use hashing for better performance)
  8. UVa 10282 - Babelfish (use `map`)
  9. UVa 10295 - Hay Points (use `map`)
  10. UVa 10374 - Election (use `map`)
  11. UVa 10686 - SQF Problem (use `map`)
  12. UVa 10815 - Andy's First Dictionary (use `set` and `string`)
  13. UVa 11062 - Andy's Second Dictionary (similar to UVa 10815, but with more twist)
  14. UVa 11136 - Hoax or what (use `map`)
  15. UVa 11239 - Open Source (use `map` and `set` to check previous strings efficiently)
  16. **UVa 11286 - Conformity *** (use `map` to keep track the frequency of a combination)
  17. UVa 11308 - Bankrupt Baker (use `map` and `set` to help manage the data)
  18. **UVa 11629 - Ballot evaluation *** (use `map`)
  19. UVa 11849 - CD (use `set` to pass the time limit, better: use hashing!)
  20. UVa 11860 - Document Analyzer (use `set` and `map`, linear scan)

---

[15] The default setting of C++ STL `priority_queue` is a Max Heap (larger to smaller) whereas the default setting of Java `PriorityQueue` is a Min Heap (smaller to larger). Tips: A Max Heap containing numbers can be easily reversed into a Min Heap (and vice versa) by inserting the negative version of those numbers. This is because the sort order of a set of positive numbers is the reverse of its negative version. This trick is used several time in this book.

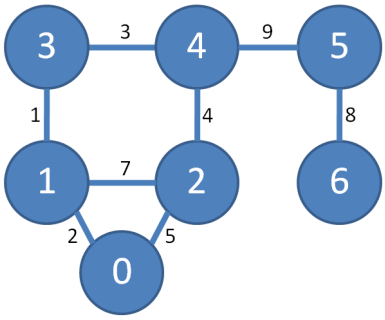
[16] This is another way to implement Kruskal's algorithm. The implementation shown in Section 4.3.2 uses `vector` + `sort` instead of `priority_queue`.

2.3 Data Structures with Our-Own Libraries

As of 1 August 2011, 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 must have bug-free implementations of these data structures ready. In this section, we discuss the ideas and the implementation examples of these data structures.

2.3.1 Graph

Graph is a pervasive data structure which appears in many Computer Science problems. Graph in its basic form is simply a collection of vertices and edges (that store connectivity information between those vertices). Later in Chapter 3, 4, and 8, we will explore many important graph problems and algorithms. To get ourselves ready, we discuss three basic ways (there are a few others) to store the information of a graph G with V vertices and E edges in this subsection.



A

	0	1	2	3	4	5	6
0		2	5				
1	2		7	1			
2	5	7			4		
3		1			3		
4			4	3		9	
5					9		8
6						8	

B

0	→ (1,2)	→ (2,5)		
1	→ (0,2)	→ (2,7)	→ (3,1)	
2	→ (0,5)	→ (1,7)	→ (4,4)	
3	→ (1,1)	→ (4,3)		
4	→ (2,4)	→ (3,3)	→ (5,9)	
5	→ (4,9)	→ (6,8)		
6	→ (5,8)			

C

w	v1	v2
1	1	3
2	0	1
3	3	4
4	2	4
5	0	2
7	1	2
8	5	6
9	4	5

Figure 2.2: Example of various Graph representations

A Adjacency Matrix, usually in the form of 2D array (see Figure 2.2.A).

In contest problems involving graph, usually the number of vertices V is known, thus we can build a 'connectivity table' by setting up a 2D static array: `int AdjMat[V][V]`. This has *space*[17] complexity of $O(V^2)$. For an unweighted graph, we set `AdjMat[i][j] = 1` if there is an edge between vertex i-j and set 0 otherwise. For a weighted graph, we set `AdjMat[i][j] = weight(i, j)` if there is an edge between vertex i-j with `weight(i, j)` and set 0 otherwise.

Adjacency Matrix is good if the connectivity between two vertices in a *small dense graph* is frequently queried. But it is not good for *large sparse graph* as it takes too much space ($O(V^2)$) and there are too many cells in the 2D array that are blank (contain zeroes). In programming contest setting, it is usually infeasible to use Adjacency Matrix when the given V is larger than 1000. Another drawback of Adjacency Matrix is that it also requires $O(V)$ time to enumerate the list of neighbors of a vertex v – an operation commonly used in many graph algorithms – even if vertex v only has a handful of neighbors. A more compact and efficient form of graph representation is the Adjacency List discussed below.

B Adjacency List, usually in the form of vector of vector pairs (see Figure 2.2.B).
Using C++ STL: `vector<vii> AdjList`, with `vii` defined as:
`typedef pair<int, int> ii; typedef vector<ii> vii; // our data type shortcuts`
Using Java API: `Vector< Vector < IntegerPair > > AdjList`.
`IntegerPair` is a simple class that contains pair of integers like `ii` above.

In Adjacency List, we have a `vector` of vector pairs. For each vertex v, we store the list of neighbors of v as 'edge information' pairs. Each 'edge information' pair contains two information, the vertex number of the neighboring vertex and the edge weight. If the graph is unweighted,

[17] In this subsection, we differentiate *space* complexity with *time* complexity. Space complexity is an asymptotic measure of space taken by a data structure. Time complexity is an asymptotic measure of time taken to run a certain algorithm (in this case, the operation on the data structure).

simply store weight = 0 (or 1) or drop this second attribute[18]. The space complexity of this graph representation is only $O(V + E)$ because if there are E bidirectional edges in the graph, this Adjacency List only stores $2E$ '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 graph, Adjacency List is more space efficient than Adjacency Matrix.

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

C Edge List, usually in the form of vector of triples (see Figure 2.2.C).
Using C++ STL: `vector< pair<int, ii> > EdgeList`.
Using Java API: `Vector< IntegerTriple > EdgeList`.
`IntegerTriple` is a simple class that contains triple of integers like `pair<int, ii>` above.

In Edge List, we store a list of all E edges, usually in some sorted order. The space complexity of this data structure is clearly $O(E)$. This graph representation is very useful for Kruskal's algorithm for MST (Section 4.3.2) where the collection of edges are sorted[19] by their length from shortest to longest. However, storing graph information in Edge List complicates many graph algorithms that work by enumerating edges that are incident to a vertex.

Example codes: `ch2_07_graph_ds.cpp`; `ch2_07_graph_ds.java`

Exercise 2.3.1.1: Show the Adjacency Matrix, Adjacency List, and Edge List of the graphs shown in Figure 4.1 (Section 4.2.1) and in Figure 4.8 (Section 4.2.9).

Exercise 2.3.1.2: Given a graph in one representation (Adjacency Matrix/AM, Adjacency List/AL, or Edge List/EL), show how to *convert* it to another representation! There are 6 possible combinations here: AM to AL, AM to EL, AL to AM, AL to EL, EL to AM, and EL to AL.

Exercise 2.3.1.3: Given a graph represented by an Adjacency Matrix, transpose it (reverse the direction of each edges). How to do the same with Adjacency List?

Implicit Graph

Some graphs do not need to be stored in a graph data structure yet we can have the complete graph information. Such graph is called *implicit* graph. You will encounter them in the subsequent chapters. There are two forms of implicit graphs:

1. The edges can be determined easily
   For example: A 2D grid. The vertices are the cells in the 2D grid. The edges can be determined easily: Two neighboring cells in the grid have an edge between them.
2. The edges can be determined with some rules
   For example: A graph contains N vertices numbered from `[0..N-1]`. There is an edge between two vertices i and j if $(i + j)$ is a prime.

2.3.2 Union-Find Disjoint Sets

Union-Find Disjoint Sets is a data structure to model a collection of *disjoint sets* which has the ability to efficiently[20] 1) 'find' which set an item belongs to (or to test whether two items belong to the same set) and 2) 'union' two disjoint sets into one bigger set.

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

[19] `pair` objects in C++ can be easily sorted. The default sorting criteria is to sort based on the first item, and if tie, based on the second item. In Java, we can write an `IntegerPair`/`IntegerTriple` class that implements `Comparable`.

[20] M operations of this data structure runs in $O(M \times \alpha(n))$. However, the inverse Ackermann function $\alpha(n)$ grows very slowly, i.e. its value is just less than 5 for any practical input size n. So, we can treat $\alpha(n)$ as constant.

These two operations are useful for Kruskal's algorithm (Section 4.3.2) or problems that involve 'partitioning', like keeping track of connected components of an undirected graph (Section 4.2.3).

These seemingly simple operations are not *efficiently* supported by the C++ STL `set` (Java `TreeSet`) which only deals with a *single* set. Having a `vector` of `sets` and looping through each one to find which set an item belongs to is expensive! C++ STL `set_union` (in `<algorithm>`) is also not efficient enough although it combines two sets in *linear time*, as we still have to deal with the shuffling of the content inside the `vector` of `sets`! We need a better data structure: The Union-Find Disjoint Sets – discussed below.

The key idea of this data structure is to keep a representative ('parent') item of each set. This information is stored in `vi pset`, where `pset[i]` tells the representative item of the set that contains item `i`. See Figure 2.3, `initSet(5)`. This function creates 5 items: {0, 1, 2, 3, 4} as 5 disjoint sets of 1 item each. Each item initially has itself as the representative.

When we want to merge two sets, we call `unionSet(i, j)` which makes both items 'i' and 'j' to have the same representative item – directly or indirectly[21] (see Path Compression below). This is done by calling `findSet(j)` – to find out the representative of item 'j', and assign that value to `pset[findSet(i)]` – to update the parent of the representative item of item 'i'. This is efficient.

See Figure 2.3, `unionSet(0, 1)`, `unionSet(1, 2)`, and `unionSet(3, 1)`. These three sub figures show what is happening when we call `unionSet(i, j)`: Each union is simply done by changing the representative item of one item to point to the other's representative item.

See Figure 2.3, `findSet(0)`. This function `findSet(i)` recursively calls itself whenever `pset[i]` is not yet itself ('i'). Then, once it finds the main representative item (e.g. 'x') for that set, it will *compress the path* by saying `pset[i] = x`. Thus subsequent calls of `findSet(i)` will be $O(1)$. This simple strategy is aptly named as 'Path Compression'.

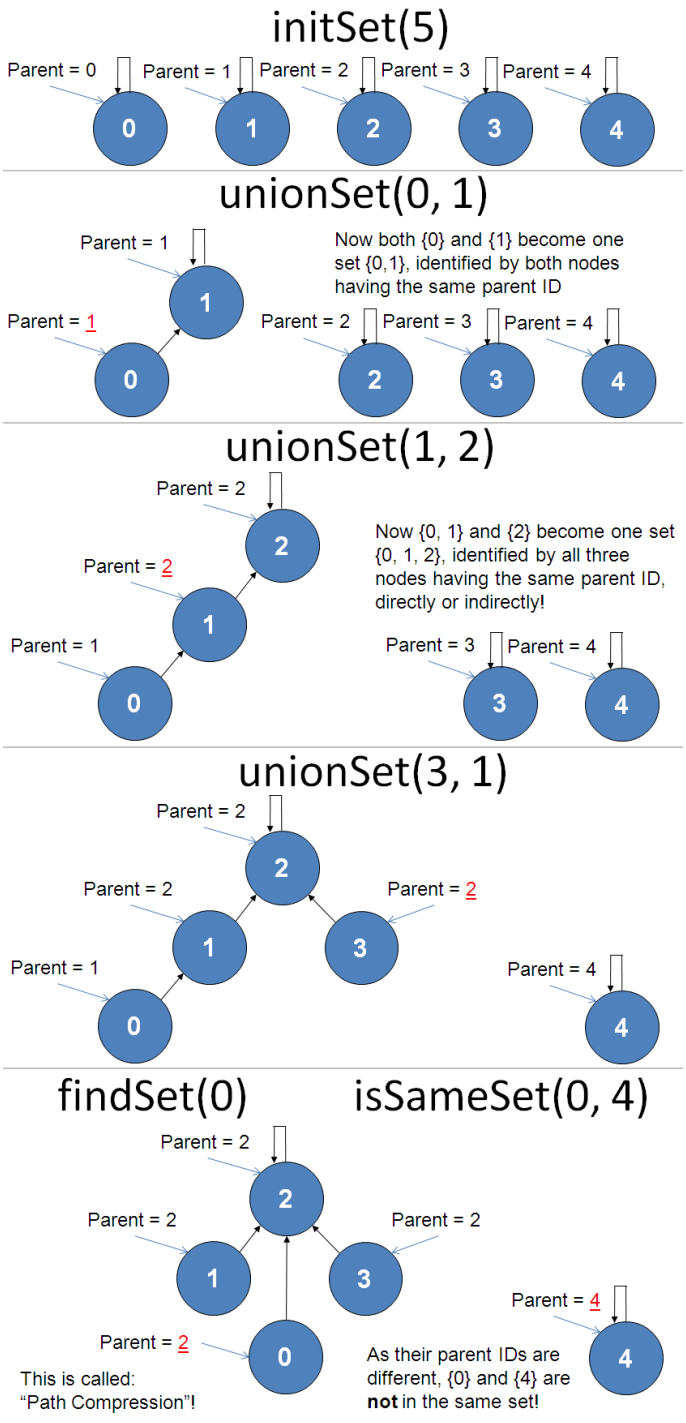


Figure 2.3: Union-Find Disjoint Sets

Figure 2.3, `isSameSet(0, 4)` shows another operation for this data structure. This function `isSameSet(i, j)` simply calls `findSet(i)` and `findSet(j)` to check if both refer to the same representative item. If yes, 'i' and 'j' belong to the same set, otherwise, they do not.

[21] See Figure 2.3. After we call `unionSet(1, 2)`, notice that parent of 1 is now 2, but parent of 0 is still 1. This will not be updated until we call `findSet(0)` later which will use the 'Path Compression' strategy.

Our library implementation for Union-Find Disjoint Sets is just these few lines of codes:

```
vi pset;                                          // remember: vi is vector<int>
void initSet(int N) { pset.assign(N, 0);
  for (int i = 0; i < N; i++) pset[i] = i; }
int findSet(int i) { return (pset[i] == i) ? i : (pset[i] = findSet(pset[i])); }
bool isSameSet(int i, int j) { return findSet(i) == findSet(j); }
void unionSet(int i, int j) { pset[findSet(i)] = findSet(j); }
```

Example codes: ch2_08_unionfind_ds.cpp; ch2_08_unionfind_ds.java

Exercise 2.3.2.1: There are two more queries that are commonly performed on the Union-Find Disjoint Sets data structure: `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`. Update the codes shown in this section to support these two queries *efficiently*!

Exercise 2.3.2.2: There is another strategy to speed up the implementation of Union-Find Disjoint Sets data structure: The 'union by rank' strategy. The idea is to give rank (upper bound of the height of the tree) to each disjoint sets. Then during `unionSet(i, j)` operation, we set the parent of the set with *fewer* members to point to the parent of the set with *more* members, thereby keeping the resulting tree 'more balanced' and thus 'shorter'. For example, in Figure 2.3, when we do `unionSet(3, 1)`, it is beneficial to set parent of vertex 3 (which is 3 itself) to point to parent of vertex 1 (which is 2) rather than setting parent of vertex 1 (which is 2) to point to parent of vertex 3 (which is 3 itself). The first approach yields a new tree of height 3 as shown in Figure 2.3, whereas the second approach yields a new tree of height 4. Do you think this heuristic will help speed up the data structure *significantly*? If yes, in which case(s)? Is there any programming trick to achieve similar effect *without* using this heuristic?

2.3.3 Segment Tree

In this subsection, we discuss a data structure which can efficiently answer *dynamic*[22] 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). 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 on the array A below, `RMQ(3, 4) = 4`, `RMQ(0, 0) = 0`, `RMQ(0, 1) = 1`, and `RMQ(0, 6) = 5`.

```
Array | Values  =   8  |  7  |  3  |  9  |  5  |  1  | 10
  A   | Indices =   0  |  1  |  2  |  3  |  4  |  5  |  6
```

There are several ways to answer this 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 is $O(n)$ per query. When n is large, such an algorithm maybe infeasible.

In this section, we answer the RMQ with Segment Tree which is another way to arrange data as a binary tree. For the array `A` above, its corresponding segment tree is shown in Figure 2.4. The root of this tree contains the full segment `[0, n-1]`. And for each segment `[L, R]`, we split them into `[L, (L+R)/2]` and `[(L+R)/2 + 1, R]` until `L = R`. See the $O(n \log n)$ `built_st` routine below. With the segment tree ready, answering an RMQ can now be done in $O(\log n)$.

The answer for `RMQ(i, i)` is trivial, which is `i` itself. But for general cases, `RMQ(i, j)`, further check is 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]`, `p2` otherwise.

For example, we want to answer `RMQ(1, 3)`. The execution in Figure 2.4 (solid lines) is as follows: Start from the root which represents segment `[0, 6]`. We know that the answer for `RMQ(1, 3)` cannot be the stored minimum value of segment `[0, 6] = 5` as it is the minimum value over a larger[23] segment `[0, 6]` than the `RMQ(1, 3)`. From the root, we have to go to the left

[22] In dynamic data structure, we frequently update and query the data, which usually make pre-processing useless.
[23] Segment `[L, R]` is said to be larger than query range `[i, j]` if `L` $\leq$ `i && j` $\leq$ `R`.

subtree as the root of the right subtree is segment `[4, 6]` which is outside[24] the `RMQ(1, 3)`.

We now move to the left subtree that represents segment `[0, 3]`. This segment `[0, 3]` is still larger than the `RMQ(1, 3)`. In fact `RMQ(1, 3)` intersects *both* the left segment `[0, 1]` and the right segment `[2, 3]` of segment `[0, 3]`, so we have to continue exploring *both* subtrees.

The left segment `[0, 1]` of `[0, 3]` is not yet inside the `RMQ(1, 3)`, so another split is necessary. From segment `[0, 1]`, we move right to segment `[1, 1]`, which is now inside[25] the `RMQ(1, 3)`. At this point, we know that `RMQ(1, 1) = 1` and we return this value to the caller. The right segment `[2, 3]` of `[0, 3]` is inside the required `RMQ(1, 3)`. From the stored minimum value inside this vertex, we know that `RMQ(2, 3) = 2`. We do *not* need to traverse further down.

Now, back in segment `[0, 3]`, we now have `p1 = RMQ(1, 1) = 1` and `p2 = RMQ(2, 3) = 2`. Because `A[p1] > A[p2]` since `A[1] = 7` and `A[2] = 3`, we now have `RMQ(1, 3) = p2 = 2`.

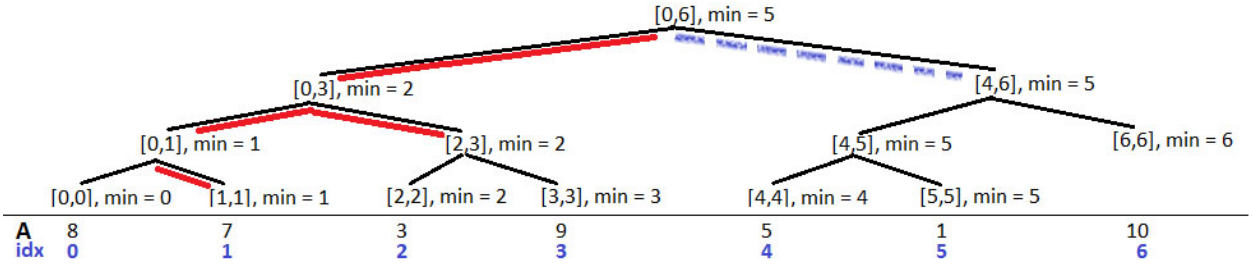


Figure 2.4: Segment Tree of Array A = {8, 7, 3, 9, 5, 1, 10}

Now let's take a look at another example: `RMQ(4, 6)`. The execution in Figure 2.4 (dashed line) is as follows: We again start from the root segment `[0, 6]`. Since it is larger than the `RMQ(4, 6)`, we move right to segment `[4, 6]`. Since this segment is exactly the `RMQ(4, 6)`, we simply return the index of minimum element that is stored in this vertex, which is 5. Thus `RMQ(4, 6) = 5`. We do not have to traverse the unnecessary parts of the tree!

In the worst case, we have *two* root-to-leaf paths which is just $O(2 \times \log n) = O(\log n)$. For example in `RMQ(3, 4) = 4`, we have one root-to-leaf path from `[0, 6]` to `[3, 3]` and another root-to-leaf path from `[0, 6]` to `[4, 4]`.

If the array `A` is static, then using Segment Tree to solve RMQ is *overkill* as there exists a Dynamic Programming (DP) solution that requires $O(n \log n)$ one-time pre-processing and $O(1)$ per RMQ. This DP solution will be discussed later in Section 3.5.2.

Segment Tree is useful if the underlying data is frequently updated (dynamic). For example, if `A[5]` is now changed from 1 to 100, then we just need to update the vertices along leaf to root path in $O(\log n)$. See path: `[5, 5]` → `[4, 5]` → `[4, 6]` → `[0, 6]` in Figure 2.5. As a comparison, DP solution requires another $O(n \log n)$ pre-processing to do the same. If we have to do such updates *many times*, using Segment Tree is a better choice.

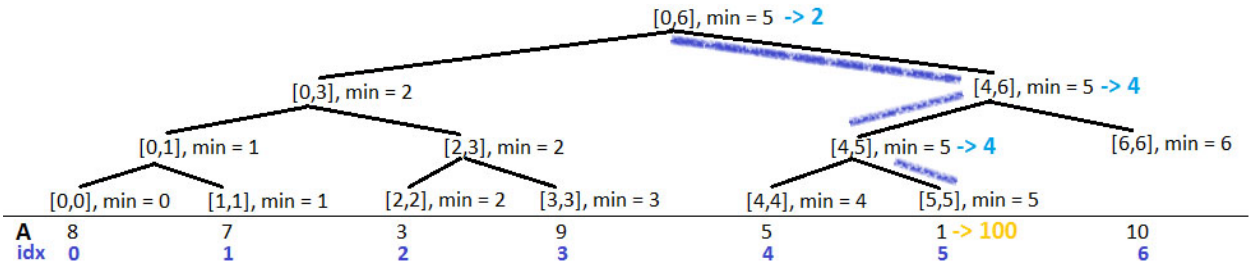


Figure 2.5: Updating Array A to {8, 7, 3, 9, 5, **100**, 10}

Our Segment Tree implementation is shown below. The code shown here supports *static* RMQ (the *dynamic* update part is left as exercise).

[24]Segment `[L, R]` is said to be outside query range `[i, j]` if `i > R || j < L`.
[25]Segment `[L, R]` is said to be inside query range `[i, j]` if `L ≥ i && R ≤ j`.

Note that there are of course other ways to implement Segment Tree, e.g. a more efficient version that only expands the segments only when needed.

```
#include <cmath>
#include <cstdio>
#include <vector>
using namespace std;
typedef vector<int> vi;

// Segment Tree Library: The segment tree is stored like a heap array
void st_build(vi &t, const vi &A, int vertex, int L, int R) {
  if (L == R)                                    // as L == R, either one is fine
    t[vertex] = L;                                          // store the index
  else {        // recursively compute the values in the left and right subtrees
    int nL = 2 * vertex, nR = 2 * vertex + 1;
    st_build(t, A, nL, L              , (L + R) / 2);      // this is O(n log n)
    st_build(t, A, nR, (L + R) / 2 + 1, R          );     // similar analysis as
    int lContent = t[nL]      , rContent = t[nR];             // with merge sort
    int lValue   = A[lContent], rValue   = A[rContent];
    t[vertex] = (lValue <= rValue) ? lContent : rContent;
} }

void st_create(vi &t, const vi &A) {             // if original array size is N,
  // the required segment tree array length is 2*2^(floor(log2(N)) + 1);
  int len = (int)(2 * pow(2.0, floor((log((double)A.size()) / log(2.0)) + 1)));
  t.assign(len, 0);   // create vector with length 'len' and fill it with zeroes
  st_build(t, A, 1, 0, (int)A.size() - 1); // recursively build the segment tree
}

int st_rmq(vi &t, const vi &A, int vertex, int L, int R, int i, int j) {
  if (i >  R || j <  L) return -1;        // current segment outside query range
  if (L >= i && R <= j) return t[vertex];  // current segment inside query range

  // compute the minimum position in the left and right part of the interval
  int p1 = st_rmq(t, A, 2 * vertex    , L              , (L + R) / 2, i, j);
  int p2 = st_rmq(t, A, 2 * vertex + 1, (L + R) / 2 + 1, R          , i, j);

  // return the position where the overall minimum is
  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; }

int st_rmq(vi &t, const vi& A, int i, int j) { // overloading, simpler arguments
  return st_rmq(t, A, 1, 0, (int)A.size() - 1, i, j); }

int main() {
  int arr[7] = { 8, 7, 3, 9, 5, 1, 10 };                  // the original array
  vi A(arr, arr + 7);
  vi st; st_create(st, A);
  printf("RMQ(1, 3) = %d\n", st_rmq(st, A, 1, 3));          // answer is index 2
  printf("RMQ(4, 6) = %d\n", st_rmq(st, A, 4, 6));          // answer is index 5
} // return 0;
```

Example codes: ch2_09_segmenttree_ds.cpp; ch2_09_segmenttree_ds.java

Exercise 2.3.3.1: Draw the Segment Tree of array `A = {10, 2, 47, 3, 7, 9, 1, 98, 21, 37}` and answer RMQ(1, 7) and RMQ(3, 8)!

Exercise 2.3.3.2: Using similar Segment Tree as in exercise 1 above, answer this Range Sum Query(i, j) (RSQ), i.e. a sum from `A[i] + A[i + 1] + ...+ A[j]`. What is `RSQ(1, 7)` and `RSQ(3, 8)`? Is this a good approach to solve this problem? (compare with Section 3.5.2).

Exercise 2.3.3.3: The Segment Tree code shown above lacks the `update` operation. Add the $O(\log n)$ `update` function to update the value of a certain index in array `A` (and its Segment Tree)!

2.3.4 Fenwick Tree

Fenwick Tree (also known as Binary Indexed Tree or BIT) was invented by *Peter M. Fenwick* in 1994 [9]. Fenwick Tree is a useful data structure for implementing *cumulative frequency tables*. For example, if we have test scores of 10 students `s = {2,4,5,5,6,6,6,7,7,8}`, then the frequency of each score `[1..8]` and their cumulative frequency are as shown in Table 2.1.

Index/ Score	Frequency	Cumulative Frequency	Short Comment Let's abbreviate Cumulative Frequency as 'CF'
0	-	-	Index 0 is ignored.
1	0	0	(CF for score 1) = 0 + 0 = 0.
2	1	1	(CF for score 2) = 0 + 0 + 1 = 1.
3	0	1	(CF for score 3) = 0 + 0 + 1 + 0 = 1.
4	1	2	(CF for score 4) = 0 + 0 + 1 + 0 + 1 = 2.
5	2	4	...
6	3	7	...
7	2	9	...
8	1	10	(CF for score 8) = (CF for score 7) + 1 = 9 + 1 = 10.

Table 2.1: Example of a Cumulative Frequency Table

This cumulative frequency table is akin to the Range Sum Query (RSQ) mentioned in Section 2.3.3 where now we perform incremental `RSQ(1, i)` $\forall i \in$ `[1..8]`. Using the example above, we have `RSQ(1, 1) = 0`, ..., `RSQ(1, 2) = 1`, ..., `RSQ(1, 6) = 7`, ..., `RSQ(1, 8) = 10`.

Rather than implementing Segment Tree to implement such cumulative frequency table, it is *far easier* to code Fenwick Tree instead (please compare the two sample source codes provided in this section versus in Section 2.3.3). Perhaps this is one of the reasons why Fenwick Tree is currently included in the IOI syllabus [10]. Fenwick Tree operations are also quite fast as they use efficient bit manipulation techniques (see Section 2.2.1).

Fenwick Trees are typically implemented as arrays (our library code uses `vector`). Fenwick Tree is a tree that is indexed by the bits of its *integer* keys. These integer keys must have a fixed range (in programming contest setting, this range can go up to `[1..1M]` – large enough for many practical problems). In the example in Table 2.1 above, the scores `[1..8]` are the integer keys.

Internally, each index is responsible for a certain range of values as shown in Figure 2.6. Let the name of the Fenwick Tree array as `ft`. Then, `ft[4]` is responsible for ranges `[1..4]`, `ft[6]` is responsible for ranges `[5..6]`, `ft[7]` is responsible for ranges `[7..7]`, etc[26].

With such arrangements, if we want to know the cumulative frequency between two indices `[1..b]`, i.e. `ft_rsq(ft, b)`, we simply add `ft[b]`, `ft[b']`, `ft[b'']`, ... until the index is 0. This sequence of indices is obtained via this bit manipulation: `b' = b - LSOne(b)`. Function `LSOne(b)` is actually `(b & (-b))`. We use the name `LSOne(b)` as with the one used in [9]. We have seen earlier in Section 2.2.1 that operation `(b & (-b))` detects the first Least Significant One in `b`. As an integer `b` only has $O(\log b)$ bits, then `ft_rsq(ft, b)` runs in $O(\log n)$ when `b = n`.

[26]In this book, we will not go into details why such arrangements work. Interested readers are advised to read [9]. Also note that we choose to follow the original implementation by [9] that skips index 0. This is to facilitate easier understanding of the bit manipulation operations later.

In Figure 2.6, the `ft_rsq(t, 6) = ft[6] + ft[4] = 5 + 2 = 7`. Notice that index 4 and 6 are responsible for range `[1..4]` and `[5..6]`, respectively. By combining them, we cover the entire range of `[1..6]`. These two indices 6, 4 are related in their binary form: `b = 6`$_{10}$ `= (110)`$_2$ can be transformed to `b' = 4`$_{10}$ `= (100)`$_2$ and subsequently to `b'' = 0`$_{10}$ `= (000)`$_2$ by stripping off the least significant one from `b` one at a time.

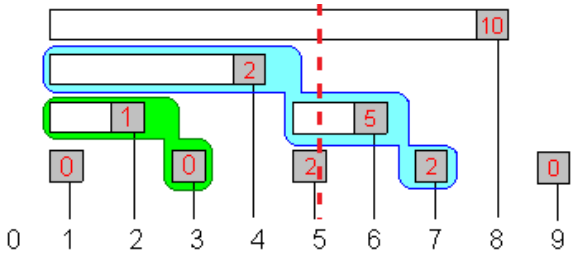


Figure 2.6: Example of a Fenwick Tree

Now, to get the cumulative frequency between two indices `[a..b]` where `a != 1` is equally simple, just do `ft_rsq(ft, b) - ft_rsq(ft, a - 1)`. In Figure 2.6, `ft_rsq(ft, 3, 6) = ft_rsq(ft, 6) - ft_rsq(ft, 2) = 7 - 1 = 6`. Again, this is just an $O(\log b) \approx O(\log n)$ operation.

Then if we want to update the value of a certain index, for example if we want to adjust the value for index `k` by a certain value `v` (note: `v` can be positive or negative), i.e. `ft_adjust(ft, k, v)`, then we have to update `ft[k]`, `ft[k']`, `ft[k'']`, ... until array index `k` exceeds the size of array. This sequence of indices are obtained via this bit manipulation: `k' = k + LSOne(k)`. Starting from any integer `k`, this operation `ft_adjust(ft, k, v)` just needs at most $O(\log n)$ steps to make `k` $\geq$ `n` where `n` is the size of the Fenwick Tree.

In Figure 2.6, `ft_adjust(ft, 5, 2)` will affect `ft` at indices `k = 5`$_{10}$ `= (101)`$_2$, `k' = 6`$_{10}$ `= (110)`$_2$, and `k'' = 8`$_{10}$ `= (1000)`$_2$. These series of indices are obtained with the bit manipulation given above. Notice that if you project a line upwards from index 5 in Figure 2.6, you will see that line indeed intersects the range handled by index 5, index 6, and index 8.

Fenwick Tree needs $O(n)$ memory and $O(\log n)$ access/update operations for a set of n integer keys. This makes Fenwick Tree an ideal data structure for solving *dynamic* RSQ problem on integer keys (the *static* RSQ problem can be solved with $O(n)$ DP + $O(1)$ per query, see Section 3.5.2).

```
#include <cstdio>
#include <vector>
using namespace std;
typedef vector<int> vi;
#define LSOne(S) (S & (-S))

void ft_create(vi &t, int n) { t.assign(n + 1, 0); }   // initially n + 1 zeroes

int ft_rsq(const vi &t, int b) {                              // returns RSQ(1, b)
  int sum = 0; for (; b; b -= LSOne(b)) sum += t[b];
  return sum; }

int ft_rsq(const vi &t, int a, int b) {                       // returns RSQ(a, b)
  return ft_rsq(t, b) - (a == 1 ? 0 : ft_rsq(t, a - 1)); }

// adjusts value of the k-th element by v (v can be +ve/inc or -ve/dec).
void ft_adjust(vi &t, int k, int v) {
  for (; k <= (int)t.size(); k += LSOne(k)) t[k] += v; }

int main() {                // idx   1 2 3 4 5 6 7 8 , index 0 is ignored!
  vi ft; ft_create(ft, 8); // ft = {0,0,0,0,0,0,0,0}
  ft_adjust(ft, 2, 1);     // ft = {0,1,0,1,0,0,0,1}, index 2,4,8 => +1
  ft_adjust(ft, 4, 1);     // ft = {0,1,0,2,0,0,0,2}, index 4,8 => +1
  ft_adjust(ft, 5, 2);     // ft = {0,1,0,2,2,2,0,4}, index 5,6,8 => +2
  ft_adjust(ft, 6, 3);     // ft = {0,1,0,2,2,5,0,7}, index 6,8 => +3
  ft_adjust(ft, 7, 2);     // ft = {0,1,0,2,2,5,2,9}, index 7,8 => +2
  ft_adjust(ft, 8, 1);     // ft = {0,1,0,2,2,5,2,10}, index 8 => +1
```

```
  printf("%d\n", ft_rsq(ft, 1, 1)); // 0 => ft[1] = 0
  printf("%d\n", ft_rsq(ft, 1, 2)); // 1 => ft[2] = 1
  printf("%d\n", ft_rsq(ft, 1, 6)); // 7 => ft[6] + ft[4] = 5 + 2 = 7
  printf("%d\n", ft_rsq(ft, 1, 8)); // 10 => ft[8]
  printf("%d\n", ft_rsq(ft, 3, 6)); // 6 => rsq(1, 6) - rsq(1, 2) = 7 - 1
} // return 0;
```

Example codes: `ch2_10_fenwicktree_ds.cpp`; `ch2_10_fenwicktree_ds.java`

Exercise 2.3.4.1: Fenwick Tree has an additional operation: Find the index with a given cumulative frequency. For example we want to know what is the minimum index/score in Table 2.1 so that there are at least 7 students covered (Answer = index/score 6). Implement this feature!

Exercise 2.3.4.2: Extend the 1D Fenwick Tree to 2D!

Programming exercises that use data structures with our own libraries:

- Graph Data Structures Problems (much more graph problems are discussed in Chapter 4)
  1. UVa 00599 - The Forrest for the Trees (apply the property of tree/forest[27])
  2. **UVa 10720 - Graph Construction *** (study 'Erdos-Gallai's theorem')
  3. **UVa 10895 - Matrix Transpose *** (transpose adjacency list)
  4. UVa 10928 - My Dear Neighbours (counting out degrees)
  5. UVa 11414 - Dreams (similar to UVa 10720; Erdos-Gallai's theorem)
  6. UVa 11550 - Demanding Dilemma (graph representation, incidence matrix)
  7. **UVa 11991 - Easy Problem from ... *** (use the idea of Adjacency List)
- Union-Find Disjoint Sets
  1. UVa 00459 - Graph Connectivity (also solvable with 'flood fill' in Section 4.2.1)
  2. **UVa 00793 - Network Connections *** (trivial; application of disjoint sets)
  3. UVa 10158 - War
  4. UVa 10178 - Count the Faces (Euler's formula for planar graph[28])
  5. UVa 10227 - Forests (merge two disjoint sets if they are consisten)
  6. **UVa 10507 - Waking up brain *** (using disjoint sets simplifies this problem)
  7. UVa 10583 - Ubiquitous Religions (count the remaining disjoint sets after unions)
  8. UVa 10608 - Friends (find the set with largest element)
  9. UVa 10685 - Nature (find the set with largest element)
  10. **UVa 11503 - Virtual Friends *** (maintain set attribute (size) in rep item)
  11. UVa 11690 - Money Matters (check if sum of money from each member of a set is 0)
  12. UVa 11966 - Galactic Bonding (brute force, check if Euclidian dist $\leq D$, union find)
- Tree-related Data Structures
  1. UVa 00297 - Quadtrees (simple quadtree problem)
  2. UVa 11297 - Census (quadtree)
  3. **UVa 11235 - Frequent Values *** (range maximum query)
  4. UVa 11350 - Stern-Brocot Tree (simple tree data structure question)
  5. **UVa 11402 - Ahoy, Pirates *** (segment tree with updates)
  6. UVa 11525 - Permutation (can use Fenwick Tree)
  7. **UVa 11926 - Multitasking *** (use FT of size 1M to indicate if a slot is free/taken)
  8. LA 2191 - Potentiometers (Dhaka06) (range sum query)
  9. LA 4108 - SKYLINE (Singapore07) (uses Segment Tree)

  Also see: Part of the solution of harder problems in Chapter 8

---

[27] Since the graph given is a forest (stated in the problem), we can use the relation between the number of edges and the number of vertices to get the number of trees and acorns. Specifically, $v - e$ = number of connected components, keep a `bitset` of size 26 to count number of vertices that has some edge. Note: Also solvable with Union-Find.

[28] For planar graph, we have Euler's formula: V - E + F = 2, where 'F' is the number of faces.

2.4 Chapter Notes

Basic data structures mentioned in Section 2.2 can be found in almost every data structure and algorithm textbooks. References to their libraries are available online at: `www.cppreference.com` and `java.sun.com/javase/6/docs/api`. Note that although these reference websites are usually given in programming contests, we suggest that you try to master the syntax of the most common library operations to save time during the actual contests!

An exception is perhaps the *lightweight set of Boolean* (a.k.a bitmask). This *unusual* data structure is not commonly taught in data structure and algorithm classes, but is quite important for competitive programmers as it allows significant speedup if used correctly. This data structure appears in various places throughout this book, e.g. in some optimized backtracking routines (Section 3.2.1), DP TSP (Section 3.5.2), DP + bitmask (Section 8.4.1). All of them use this data structure instead of `vector<boolean>` or `bitset<size>` mainly because of its efficiency. Interested reader is encouraged to read a book: "Hacker's Delight" [41] that discusses bit manipulation.

Extra references for data structures mentioned in Section 2.3 are as follows: For Graph data structure, see [32] and Chapter 22-26 of [3]. For Union-Find Disjoint Sets, see Chapter 21 of [3]. For Segment Tree and other geometric data structures, see [5]. For Fenwick Tree, see [17].

With more experience and by looking at our sample codes, you will master more tricks in using these data structures. Please spend some time to explore the sample codes listed in this book. They are available at: `https://sites.google.com/site/stevenhalim/home/material`.

There are few more data structure discussed in this book: The string-specific data structures (**Suffix Trie/Tree/Array**) 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 contest, please study and master data structures beyond what we present in this book. For example, **AVL Tree**, **Red Black Tree**, or **Splay Tree** are useful for certain contest problems where you need to implement and augment (add more data) a balanced BST; **Interval Tree** which is similar to Segment Tree, **Quad Tree** for partitioning 2D space; etc.

Notice that many of the efficient data structures shown in this book have the spirit of Divide and Conquer (discussed in Section 3.3).

There are ≈ **117 UVa (+ 7 others) programming exercises** discussed in this chapter.
(Only 43 in the first edition, a 188% increase).

There are **18 pages** in this chapter.
(Only 12 in the first edition, a 50% increase).

Profile of Data Structure Inventors

Rudolf Bayer (born 1939) has been Professor (emeritus) of Informatics at the Technical University of Munich. He invented the Red-Black (RB) tree which is typically used in C++ STL `map` and `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 AVL tree is the abbreviation of the two inventors: Adelson-Velskii and Landis himself.

Peter M. Fenwick is a Honorary Associate Professor in The University of Auckland. He invented Binary Indexed Tree in 1994 with the original purpose for "cumulative frequency tables of arithmetic compression" [9]. It has since evolved into a programming contest material for efficient yet easy to implement data structure by its inclusion in the IOI syllabus [10].

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 highlight four problem solving paradigms commonly used to attack problems in programming contests, namely Complete Search, Divide & Conquer, Greedy, and Dynamic Programming. Both IOI and ICPC contestants need to master all these problem solving paradigms so that they can attack the given problem with the appropriate 'tool', rather than 'hammering' every problem with the brute-force solution (which is clearly not competitive).

Our advice before you start reading: Do not just remember the solutions for the problems discussed in this chapter, but remember the way, the spirit of solving those problems!

3.2 Complete Search

Complete Search, also known as brute force or recursive backtracking, is a method for solving a problem by searching (up to) the entire search space to obtain the required solution.

In programming contests, a contestant *should* develop a Complete Search solution when there is clearly no clever algorithm available (e.g. the problem of enumerating *all* permutations of $\{0, 1, 2, \ldots, N-1\}$, which clearly requires $O(N!)$ operations) or when such clever algorithms exist, but overkill, as the input size happens to be small (e.g. the problem of answering Range Minimum Query as in Section 2.3.3 but on a static array with $N \leq 100$ – solvable with an $O(N)$ loop).

In ICPC, Complete Search should be the first solution to be considered as it is usually easy to come up with the solution and to code/debug it. Remember the 'KISS' principle: Keep It Short and Simple. A *bug-free* Complete Search solution should *never* receive 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. Thus a Complete Search solution may receive a Time Limit Exceeded (TLE) verdict. With proper analysis, you can determine which is the likely outcome (TLE versus AC) before attempting to code anything (Table 1.4 in Section 1.2.2 is a good gauge). If Complete Search can likely pass the time limit, then go ahead. This will then give you more time to work on the harder problems where Complete Search is too slow.

In IOI, we usually need better problem solving techniques as Complete Search solutions are usually only rewarded with low marks. Nevertheless, Complete Search should be used when we cannot come up with a better solution in order to grab that precious low marks.

Sometimes, running Complete Search on *small instances* of a challenging problem can give us some patterns from the output (it is possible to visualize the pattern for some problems) that can be exploited to design a faster algorithm. You may want to see some combinatorics problems in Section 5.4 that can be solved this way. Then, the Complete Search solution will also act as a verifier on *small instances* for the faster but non-trivial algorithm that you manage to develop.

In this section, we give two examples of this simple yet can be challenging paradigm and provide a few tips to give Complete Search solution a better chance to pass the required Time Limit.

3.2.1 Examples

We show two examples of Complete Search: One that is implemented iteratively and one that is implemented recursively (backtracking). We will also discuss a few algorithm design choices to make some 'impossible' cases become possible.

Iterative Complete Search: UVa 725 - Division

Abridged problem statement: Find and display all pairs of 5-digit numbers that between them 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`.

A quick analysis shows that `fghij` can only range from `01234` to `98765`, which is $\approx 100K$ possibilities. For each tried `fghij`, we can get `abcde` from `fghij * N` and then check if all digits are different. $100K$ operations are small. Thus, iterative Complete Search is feasible.

Exercise 3.2.1.1: What is the advantage of iterating through `fghij` and not through `abcde`?

Exercise 3.2.1.2: Does a 10! algorithm that permutes `abcdefghij` work for this problem?

Exercise 3.2.1.3: Checking `fghij` from `01234` to `98765` is actually 'overkill'. There is a smaller safe upperbound value than `98765`. What is the better upperbound value?

Exercise 3.2.1.4: Solve UVa 471 - Magic Numbers which is very similar to this example!

Recursive Backtracking: UVa 750 - 8 Queens Chess Problem

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

A naïve solution tries all $8^8 \approx 17M$ possible arrangements of 8 queens in an 8×8 board, putting each queen in each possible cell and filter the invalid ones. Complete Search like this receives Time Limit Exceeded (TLE) response. We should reduce the search space by pruning the obviously incorrect board configurations!

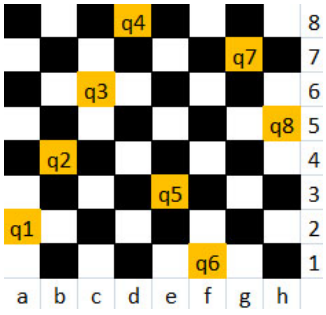


Figure 3.1: 8-Queens

We know that no two queens can share the same column, thus we can simplify the original problem to a problem of finding valid permutation of 8! row positions; `row[i]` stores the row position of the queen in column i. Example: `row = {2, 4, 6, 8, 3, 1, 7, 5}` as in Figure 3.1 is one of the solution for this problem; `row[1] = 2` implies that queen in column 1 (labeled as 'a' in Figure 3.1) is placed in row 2, and so on (the index starts from 1 in this example). Modeled this way, the search space goes *down* from $8^8 = 17M$ to $8! = 40K$.

We also know that no two queens can share any of the two diagonal lines. Assuming queen A is at `(i, j)` and queen B is at `(k, l)`, then 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. In other words, queen A and B lie on one of each other's two diagonal lines.

A recursive backtracking solution places the queens one by one from column 1 to 8, obeying the two constraints above. Finally, if a candidate solution is found, check if one of the queen satisfies the input constraint, i.e. `row[b] == a`. This solution is Accepted.

We provide our code in the next page below. If you have never coded a recursive backtracking solution before, please scrutinize it and perhaps re-code it using your own coding style.

```
/* 8 Queens Chess Problem */
#include <cmath>
#include <cstdio>
#include <cstring>
using namespace std;

int row[9], TC, a, b, lineCounter;                 // ok to use global variables

bool place(int col, int tryrow) {
  for (int prev = 1; prev < col; prev++)       // check previously placed queens
    if (row[prev] == tryrow || (abs(row[prev] - tryrow) == abs(prev - col)))
      return false; // an infeasible solution if share same row or same diagonal
  return true;
}

void backtrack(int col) {
  for (int tryrow = 1; tryrow <= 8; tryrow++)            // try all possible row
    if (place(col, tryrow)) {     // if can place a queen at this col and row...
      row[col] = tryrow;                    // put this queen in this col and row
      if (col == 8 && row[b] == a) { // a candidate solution, (a, b) has 1 queen
        printf("%2d      %d", ++lineCounter, row[1]);
        for (int j = 2; j <= 8; j++) printf(" %d", row[j]);
        printf("\n"); }
      else backtrack(col + 1);                     // recursively try next column
}   }

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

Example codes: ch3_01_UVa750.cpp; ch3_01_UVa750.java

Exercise 3.2.1.4: The code shown here can be further optimized a little bit more by pruning the search when '`row[b] != a`' during the recursion. Modify it!

Exercise 3.2.1.5: Solve similar problems (UVa 167 and UVa 11085) using the code shown here!

Exercise 3.2.1.6: For a more challenging version of this problem, attempt UVa 11195 - Another n-Queen Problem. On top of the technique mentioned in this section, you also need to use efficient data structure for checking if a certain left/right diagonal can still be used (there are $2N - 1$ left diagonals in an $N \times N$ board, similarly for right diagonals). To further speed up the solution, use the bitmask technique discussed in Section 2.2.1.

3.2.2 Tips

The biggest gamble in writing a Complete Search solution is whether it will be able to pass the time limit. If the time limit is 1 minute and your program currently runs in 1 minute 5 seconds,

you may want to tweak the 'critical code'[1] of your program first instead of painfully redoing the problem with a faster algorithm – which may not be trivial to design.

Here are some tips that you may want to consider when designing your solution, especially a Complete Search solution, to give it a higher chance of passing the Time Limit.

Tip 1: Filtering versus Generating

Programs that examine lots (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 8^8 time complexity and the iterative solution for UVa 725. 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 8-queens solver with 8! complexity plus diagonal checks. Usually 'generator' programs are written recursively.

Generally, filters are easier to code but run slower. Do the math to see if a filter is good enough or if you need to create a generator.

Tip 2: Prune Infeasible Search Space Early

In generating solutions using recursive backtracking (see tip 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. For example, see the diagonal check in 8-queens solution above. Suppose we have placed a queen at `row[1] = 2`, then placing another queen at `row[2] = 1` or `row[2] = 3` will cause a diagonal conflict and placing another queen at `row[2] = 2` will cause a row conflict. Continuing from any of these partial solutions will never lead to a valid solution. Thus we can prune these partial solutions right at this juncture, concentrate on only valid positions of `row[2] = {4, 5, 6, 7, 8}`, thus saving overall runtime.

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 this 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, 5, 3, 1, 6, 8, 2, 4}` is the horizontal reflection of Figure 3.1.

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

Sometimes it is helpful to generate tables or other data structures that enable the fastest possible 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 the more recent programming contest problems.

Example using the 8 queens chess problem above: Since we know that there are only 92 solutions, we can create a 2D array `int solution[92][8]` and then fill it with all 92 valid permutations of 8 queens row positions! That is, we create a generator program (which takes some runtime) to fill this 2D array `solution`. But afterwards, we write *another* program that just prints out the correct permutations out of these 92 that has one queen at coordinate (a, b) (very fast).

Tip 5: Try Solving the Problem Backwards

Surprisingly, some contest problems look far easier when they are solved 'backwards' (from the *less obvious* angle) than when they are solved using a frontal attack (from the more obvious angle). Be ready to process the data in some order other than the obvious.

[1]It is said that every program is doing most of its task in only about 10% of the code – the critical code.

This tip is best shown using an example: UVa 10360 - Rat Attack: Imagine a 2D array (up to 1024×1024) containing rats. There are $n \leq 20000$ rats at some cells, determine which cell `(x, y)` that 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.

The first option is to attack this problem frontally: Try bombing each of the 1024^2 cells and see which one is the most effective. For each bombed cell `(x, y)`, we need to do $O(d^2)$ scans 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. This is TLE!

The second option is to attack this problem backwards: Create an array `int killed[1024][1024]`. For each rat population at coordinate `(x, y)`, add the value of array `killed[i][j]` with the number of rats in `(x, y)` for all `(i, j)` within the square-bombing radius (i.e. $|i - x| \leq d$ and $|j - y| \leq d$). This is because all these rats will be killed if a bomb is placed in coordinate `(i, j)`. This pre-processing takes $O(n \times d^2)$ operations. Then, to determine the most optimal bombing position, we find the coordinate of the highest entry in array `killed`, which can be done in 1024^2 operations. This backwards 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 AC.

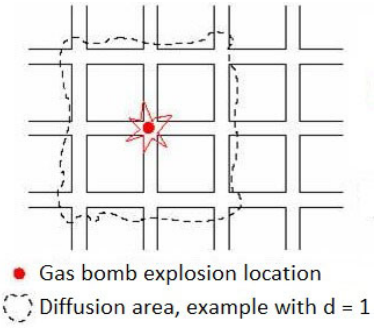


Figure 3.2: UVa 10360 [28]

Tip 6: Optimizing Your Source Code

There are many tricks that you can use to optimize your code. Understanding computer hardware, especially I/O, memory, and cache behavior, can help you design a better program. Some examples:

1. Use the faster C-style `scanf/printf` rather than `cin/cout`. For Java user, `BufferedReader/BufferedWriter` are faster. However, we decide to stick with `Scanner/System.out.println` in this book for their useful `NextBigInteger()` and `printf()` functions.
2. Use the *expected* $O(n \log n)$ but cache-friendly quicksort (inside C++ STL `algorithm::sort` as part of 'introsort') rather than the true $O(n \log n)$ but not cache-friendly heapsort (as its root-to-leaf/leaf-to-root operations actually span wide range of indices – lots of cache miss).
3. Access a 2D array in a row major fashion (row by row) rather than column by column.
4. Bit manipulation on integer is more efficient than index manipulation of an array of boolean (see the lightweight set of Boolean in Section 2.2.1). If we need more than 64 bits, use C++ STL `bitset` rather than `vector<bool>` (e.g. for Sieve of Eratosthenes in Section 5.5.1).
5. Use lower level data structure/type whenever possible. For example, use `array` with size equals to the maximum size of input instead of using resizable `vector`; use 32-bit `int` instead of 64-bit `long long` as the 32-bit `int` is faster in most 32-bit online judge systems.
6. Declare a bulky data structure (e.g. large array) just once by setting it to have global scope. Allocate enough memory to deal with the largest input of the problem. This way, we do not have to pass the bulky data structure as function arguments. For multiple-input problem, simply clear/reset the content of the data structure before dealing with the next test case.
7. In C/C++, *appropriate* usage of macros or inline functions can reduce running time.
8. For C++ user, using C-style character array is faster than using C++ STL string. For Java user, be careful with `String` manipulation as Java `String` objects are immutable. Operations on Java `String` can thus be very slow. Use Java `StringBuffer` instead.
9. For Java user, use the faster `ArrayList` rather than `Vector`. Java `Vector` is thread safe but this is not needed in competitive programming.

Browse the Internet or reference books (e.g. [41]) to find 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 gives you faster runtime.

Tip 7: Use Better Data Structures & Algorithms :)

No kidding. Using better data structures and algorithms always outperforms any optimization tips mentioned in Tips 1-6 above. If all else above fails, abandon Complete Search approach.

Remarks About Complete Search in Programming Contests

If a problem is decided to be solvable by Complete Search, it will also be clear when to use iterative or recursive backtracking approach. Iterative approach is used when one can derive the different states *easily* with some formula relative to a certain *counter* and all states have to be checked, e.g. scanning all indices of array, enumerating all possible subsets of a small set. Recursive Backtracking is used when it is hard to derive the different states with some simple counter and/or one also wants to prune the search space, e.g. the 8 queens chess problem. Another thing to note is if the search space of a problem that is solvable with Complete Search is large, the recursive backtracking that allows early pruning of infeasible search space is usually used.

The best way to improve your Complete Search skills is to solve many problems solvable with Complete Search. We have categorized a lot of such problems into four sub-categories below. Please attempt as many as possible, especially those that are highlighted as **must try ***.

Programming Exercises solvable using Complete Search:

- Iterative (The Easier Ones)
  1. UVa 00102 - Ecological Bin Packing (try all 6 possible combinations)
  2. UVa 00105 - The Skyline Problem (height map)
  3. UVa 00140 - Bandwidth (max n is just 8, use `next_permutation`[2])
  4. UVa 00154 - Recycling (try all combinations)
  5. UVa 00188 - Perfect Hash (try all)
  6. UVa 00256 - Quirksome Squares (brute force, math, pre-calculate-able)
  7. UVa 00296 - Safebreaker (try all 10000 possible codes)
  8. UVa 00331 - Mapping the Swaps ($n \leq 5$...)
  9. UVa 00347 - Run, Run, Runaround Numbers (simulate the process)
  10. UVa 00386 - Perfect Cubes (use 4 nested loops with pruning)
  11. **UVa 00435 - Block Voting *** (only 2^{20} possible coalition combinations)
  12. UVa 00441 - Lotto (use 6 nested loops)
  13. UVa 00471 - Magic Numbers (somewhat similar to UVa 725)
  14. UVa 00617 - Nonstop Travel (try all integer speeds from 30 to 60 mph)
  15. UVa 00626 - Ecosystem (use 3 nested loops)
  16. UVa 00725 - Division (elaborated in this section)
  17. UVa 00927 - Integer Sequence from ... (use sum of arithmetic series; brute force)
  18. UVa 10041 - Vito's Family (input is not big; try all possible location of Vito's House)
  19. UVa 10102 - The Path in the Colored Field (simple 4 nested loops will do[3])
  20. UVa 10365 - Blocks (use 3 nested loops with pruning)
  21. UVa 10487 - Closest Sums (sort, and then do $O(n^2)$ pairings)
  22. UVa 10662 - The Wedding (3 nested loops!)
  23. UVa 10677 - Base Equality (try all from r2 to r1)
  24. UVa 10976 - Fractions Again ? (total solutions is asked upfront; do brute force twice)

[2]The algorithm inside `next_permutation` is recursive, but codes that use it usually do not look recursive.
[3]Get maximum of minimum Manhattan distance from a '1' to a '3'. 4 nested loops is OK, we do not need BFS.

25. UVa 11001 - Necklace (brute force math, maximize function)
26. UVa 11005 - Cheapest Base (try all possible bases from 2 to 36)
27. UVa 11059 - Maximum Product (input is small)
28. UVa 11078 - Open Credit System (one linear scan)
29. **UVa 11242 - Tour de France *** (iterative complete search + sorting)
30. UVa 11342 - Three-square (with pre-calculation)
31. UVa 11412 - Dig the Holes (use `next_permutation`, find one possibility from 6!)
32. UVa 11565 - Simple Equations (3 nested loops with pruning)
33. **UVa 11742 - Social Constraints *** (use `next_permutation`, filter 8! possibilities)
34. LA 4142 - Expert Enough (Jakarta08)
35. LA 4843 - Sales (Daejeon10)

- Iterative (The More Challenging Ones)

1. UVa 00253 - Cube painting (try all, similar problem in UVa 11959)
2. UVa 00639 - Don't Get Rooked (generate 2^{16} combinations, prune invalid ones)
3. UVa 00703 - Triple Ties: The Organizer's ... (use 3 nested loops)
4. UVa 00735 - Dart-a-Mania (3 nested loops is sufficient to solve this problem)
5. UVa 00932 - Checking the N-Queens ... (brute force; move N queens to 8 dirs)
6. UVa 10125 - Sumsets (sort; brute force; reduce complexity with binary search)
7. UVa 10177 - (2/3/4)-D Sqr/Rects/Cubes/... (use 2/3/4 nested loops, precalc)
8. UVa 10360 - Rat Attack (this problem is also solvable using 1024^2 DP max sum)
9. **UVa 10660 - Citizen attention offices *** (7 nested loops! manhattan distance)
10. UVa 10973 - Triangle Counting ($O(n^3)$, 3 nested loops with pruning)
11. UVa 11108 - Tautology (try all $2^5 = 32$ values, using recursive parsing)
12. **UVa 11205 - The Broken Pedometer *** (try all 2^{15} bitmask)
13. **UVa 11553 - Grid Game *** (solvable with brute force (or DP + bitmask but overkill))
14. UVa 11804 - Argentina (5 nested loops!)
15. UVa 11959 - Dice (try all possible dice positions, compare with the 2nd one)

- Recursive Backtracking (The Easier Ones)

1. UVa 00167 - The Sultan Successor (8 queens chess problem)
2. UVa 00222 - Budget Travel (input is not large)
3. UVa 00380 - Call Forwarding (backtracking, work with strings)
4. UVa 00487 - Boggle Blitz (use `map`)
5. UVa 00524 - Prime Ring Problem (also see Section 5.5.1)
6. UVa 00539 - The Settlers of Catan (longest simple path in *small* general graph)
7. UVa 00574 - Sum It Up (you are asked to print all solutions with backtracking)
8. UVa 00598 - Bundling Newspaper (you have to print all solutions with backtracking)
9. **UVa 00624 - CD *** (input size is small, backtracking; also ok with DP-subset sum)
10. UVa 00628 - Passwords (backtracking)
11. UVa 00677 - All Walks of length "n" ... (print all solutions with backtracking)
12. UVa 00729 - The Hamming Distance ... (backtracking)
13. UVa 00750 - 8 Queens Chess Problem (solution already shown in this section)
14. UVa 10017 - The Never Ending Towers ... (classical problem)
15. UVa 10276 - Hanoi Tower Troubles Again (insert a number one by one; backtracking)
16. UVa 10344 - 23 Out of 5 (you can rearrange the 5 operands and the 3 operators)
17. UVa 10452 - Marcus, help (at each pos, Indy can go forth/left/right; try all)
18. UVa 10475 - Help the Leaders (generate and prune; try all)
19. **UVa 10503 - The dominoes solitaire *** (max 13 spaces only; backtracking)
20. UVa 10576 - Y2K Accounting Bug (generate all, prune invalid ones, take the max)
21. **UVa 11085 - Back to the 8-Queens *** (similar to UVa 750, use pre-calc)
22. UVa 11201 - The Problem with the Crazy ... (backtrack involving string)
23. LA 4844 - String Popping (Daejeon10)

24. LA 4994 - Overlapping Scenes (KualaLumpur10)

- Recursive Backtracking (The More Challenging Ones)
  1. UVa 00165 - Stamps (requires some DP too)
  2. UVa 00193 - Graph Coloring (Max Independent Set, input is small)
  3. UVa 00208 - Firetruck (backtracking with some pruning)
  4. **UVa 00416 - LED Test *** (backtrack, try all)
  5. UVa 00433 - Bank (Not Quite O.C.R.) (similar to UVa 416)
  6. UVa 00565 - Pizza Anyone? (backtracking with lots of pruning)
  7. UVa 00868 - Numerical maze (backtrack: row 1 to row N; 4 ways; several constraints)
  8. UVa 10094 - Place the Guards (like n-queens problem, but must find/use the pattern!)
  9. **UVa 10309 - Turn the Lights Off *** (brute force first row in 2^{10}, the rest follows)
  10. UVa 10582 - ASCII Labyrinth (convert complex input to simpler one; then backtrack)
  11. **UVa 11195 - Another n-Queen Problem *** (see **Exercise 3.2.1.6**)
  12. LA 4793 - Robots on Ice (World Finals Harbin10, recommended problem)
  13. LA 4845 - Password (Daejeon10)

ACM ICPC World Finals 2010 - Problem I - Robots on Ice

Problem I - 'Robots on Ice' in the recent ACM ICPC World Finals 2010 can be viewed as a 'tough test on pruning strategy'. The problem is simple: Given an $M \times N$ board with 3 check-in points {A, B, C}, find a Hamiltonian[4] path of length $(M \times N)$ from coordinate (0, 0) to coordinate (0, 1). This Hamiltonian path must hit check point {A, B, C} at one-fourth, one-half, and three-fourths of the way through its path, respectively. Constraints: $2 \leq M, N \leq 8$.

Example: If given the following 3×6 board with A = (row, col) = (2, 1), B = (2, 4), and C = (0, 4) as in Figure 3.3, then we have two possible paths.

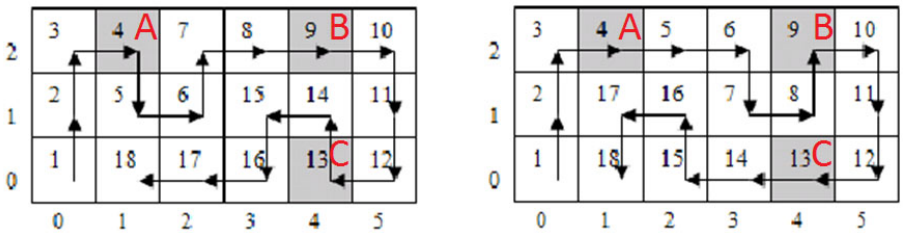


Figure 3.3: Visualization of LA 4793 - Robots on Ice

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 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). 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!

These five pruning strategies are sufficient to solve LA 4793.

Example codes: `ch3_02_LA4793.cpp`; `ch3_02_LA4793.java`

[4] A Hamiltonian path is a path in an undirected graph that visits each vertex exactly once.

3.3 Divide and Conquer

Divide and Conquer (abbreviated as D&C) is a problem solving paradigm where we try to make a problem *simpler* by 'dividing' it into smaller parts and 'conquering' them. 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 produce a complete solution for the main problem.

We have seen this D&C paradigm in previous sections in this book: Various sorting algorithms (e.g. Quick Sort, Merge Sort, Heap Sort) and Binary Search in Section 2.2.1 utilize this paradigm. The way data is organized in Binary Search Tree, Heap, Segment Tree, and Fenwick Tree in Section 2.2.2, 2.3.3, and 2.3.4 also has the spirit of Divide & Conquer.

3.3.1 Interesting Usages of Binary Search

In this section, we discuss the spirit of D&C paradigm around a well-known Binary Search algorithm. We still classify Binary Search as 'Divide' and Conquer paradigm although some references (e.g. [24]) suggest that it should be 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 other ways than its ordinary usage.

Binary Search: The Ordinary Usage

Recall: The *ordinary* usage of Binary Search is for searching an item in a *static sorted array*. We check the middle portion of the sorted array if it contains what we are looking for. If it is or there is no more item to search, we stop. Otherwise, we decide whether the answer is on the left or right portion of the sorted array. As the size of search space is halved (binary) after each check, the complexity of this algorithm is $O(\log n)$. In Section 2.2.1, we have seen that this algorithm has library routines, e.g. C++ STL `algorithm::lower_bound` (Java `Collections.binarySearch`).

This is *not* the only way to use binary search. The pre-requisite to run binary search algorithm – a *static sorted array (or vector)* – can also be found in other uncommon data structures, as in the root-to-leaf path of a tree (not necessarily binary nor complete) that has *min heap* property.

Binary Search on Uncommon Data Structures

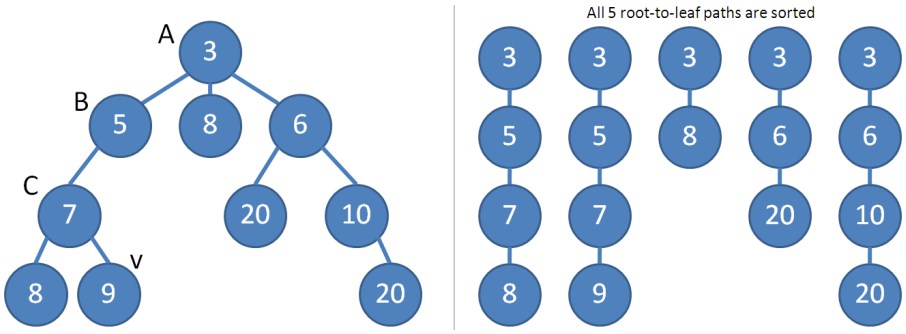


Figure 3.4: My Ancestor

This original problem is titled 'My Ancestor' and used in Thailand ICPC National Contest 2009. Brief 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 3.4 (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 value ≥ 4. If $P = 7$, then the answer is the vertex labeled with 'C' with value 7. If $P \geq 9$, then there is no answer.

The naïve solution is to do the following linear $O(N)$ scan per query: Start from the given vertex v, then move up the family tree until we hit the first vertex which direct parent has value $< P$ or until we hit the root. If this vertex is not vertex v itself, we have found the solution. In overall, as there are Q queries, this approach runs in $O(QN)$ (the input tree can be a sorted linked list of length N) and will get TLE as $N \leq 80K$ and $Q \leq 20K$.

A better solution is to store all the $20K$ queries first (we do not have to answer immediately). Then traverse the family tree *just once* starting from root using $O(N)$ preorder tree traversal algorithm (Section 4.7.2). This preorder tree traversal is slightly modified to remember the partial root-to-current-vertex *sorted* array as it goes. The array is sorted because the vertices along the root-to-current-vertex path have increasing weights, see Figure 3.4 (right). The preorder tree traversal on the tree shown in Figure 3.4 (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 tree traversal, when we are at a vertex that is asked in the query, we perform a $O(\log N)$ **binary search** (to be precise: `lower_bound`) on that partial root-to-current-vertex sorted array to get the ancestor closest to root with value at least P and record the solution. Finally, do an $O(Q)$ post-processing to output the results. The overall time complexity of this approach is $O(Q \log N)$, which is now manageable.

Bisection Method

What we have seen so far are the usages of binary search to find items in a static sorted array. However, the binary search **principle**[5] can also be used to find the root of a function that may be difficult to compute mathematically.

Example problem: You want to buy a car with loan and 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 $i\%$ interest rate for any unpaid loan at the end of each month. What is the amount of money d that you must pay per month (rounded to 2 digits after decimal point)?

Suppose $d = 576.19$, $m = 2$, $v = 1000$, and $i = 10\%$. After one month, your loan becomes $1000 \times (1.1) - 576.19 = 523.81$. After two months, your loan becomes $523.81 \times (1.1) - 576.19 \approx 0$. But if we are only given $m = 2$, $v = 1000$, and $i = 10\%$, how to determine that $d = 576.19$? In another words, find the root d such that loan payment function $f(d, m, v, i) \approx 0$.

The *easy* way to solve this root finding problem is to run the bisection method. We pick a reasonable range as the starting point. In this case, we want to find d within range `[a..b]`. $a = 1$ as we have to pay something (at least $d = 1$ dollar). $b = (1 + i) \times v$ 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, $b = (1 + 0.1) \times 1000 = 1100$ dollars. On this example, bisection method runs as in Table 3.1.

a	b	d = (a+b)/2	status	action
1	1100	550.5	undershoot by 53.95	increase d
550.5	1100	825.25	overshoot by 523.025	decrease d
550.5	825.25	687.875	overshoot by 234.5375	decrease d
550.5	687.875	619.1875	overshoot by 90.29375	decrease d
550.5	619.1875	584.84375	overshoot by 18.171875	decrease d
550.5	584.84375	567.671875	undershoot by 17.8890625	increase d
...	...	...	a **few** iterations later ...	...
...	...	576.190476...	stop here; the error is now less than ϵ	answer = 576.19

Table 3.1: Running Bisection Method on the Example Function

[5]We use the term 'binary search principle' as a Divide and Conquer technique that involves halving the range of the possible answer. We use the term 'binary search algorithm' (finding index of certain item in sorted array), 'bisection method' (finding the root of a function), and 'binary search the answer' (discussed next) as the instances of this principle.

For bisection method to work[6], 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 in the example problem above). Bisection method only requires $O(\log_2((b-a)/\epsilon))$ iterations to get an answer that is good enough (the error is smaller than a small constant ϵ that we can tolerate). In this example, bisection method only takes $\log_2 1099/\epsilon$ tries. Using a small $\epsilon =$ `1e-9`, this is just ≈ 40 tries. Even if we use an even smaller $\epsilon =$ `1e-15`, we still just need ≈ 60 tries. Bisection method is much more efficient compared to linearly trying each possible value of $d =$`[1..1100]`$/\epsilon$.

Binary Search the Answer

The abridged version of UVa 11935 - Through the Desert is as follow: 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 like 'drive (that consumes fuel)', 'experience gas leak (that further reduces the amount of fuel left)', 'encounter gas station (that allows you to refuel up to the original capacity of your jeep's fuel tank)', 'encounter mechanic (fixes all leaks)', or 'reach goal (done)'. Now upon reaching the other side of the desert, you wonder what is the *smallest possible* fuel tank capacity so that your jeep is able to reach the goal event. The answer should be precise with three digits after the decimal point.

If we know the jeep's fuel tank capacity, then this problem is just a simulation problem. From the starting side, simulate each event one by one. See if you can reach the goal event without running out of fuel. The problem is that we do not know the jeep's fuel tank capacity – this value is actually the answer that we are looking for in this problem.

From the problem description, we can compute that the range of possible answer is between `[0.000..10000.000]`. However, there are $10M$ such possibilities. Trying each value one by one will get a TLE verdict.

Fortunately, there is a property that we can exploit. Suppose that the correct answer is X. Then setting your jeep's fuel tank capacity to any value between `[0.000..X-0.001]` 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 outline to get the solution for this problem.

```
#define EPS 1e-9   // this epsilon can be adjusted; usually 1e-9 is small enough

bool can(double f) {                    // details of this simulation is omitted
  // return true if your 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 above is over, we have the answer
```

Note that some programmers choose to use 'for loop' instead of 'while loop' to avoid precision error when testing `fabs(hi - lo)` $>$ `EPS` and thus trapped in an infinite loop.

[6]Note that the requirement of bisection method (which uses binary search principle) is slightly different from the more well-known binary search algorithm which needs a sorted array.

The only changes 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: If you are new with this 'binary search the answer' technique, solve this UVa 11935 and UVa 11413 - Fill the Containers (binary search on integer range).

Remarks About Divide and Conquer in Programming Contests

Other than its natural usage in many advanced data structures, the most commonly used form of 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 its various form of usages.

Once you are more familiar with the 'Binary Search the Answer' technique discussed in this section, please explore Section 8.2.1 to see few more programming exercises that use this technique with other algorithm that we will discuss in the next few chapters.

Another 'variant' of binary search is the *ternary* search for finding the maximum (or minimum) of a *unimodal* function (that is, strictly increasing and then strictly decreasing; or vice versa for finding the minimum). While binary search divides the range into two and decides which half to continue, ternary search divides the range into three and decide which two-thirds to continue. However, ternary search is rarely used in programming contests.

We have noticed that there are not that many Divide and Conquer problems outside the binary search category. Most D&C solutions are 'geometric-related', 'problem specific', and thus cannot be mentioned one by one in this book. We will revisit some of them in Section 7.4.

Programming Exercises solvable using Binary Search:

1. UVa 00679 - Dropping Balls (like binary search; bit manipulation solution exists)
2. UVa 00957 - Popes (complete search + binary search: `upper_bound`)
3. UVa 10077 - The Stern-Brocot Number ... (binary search)
4. UVa 10341 - Solve It (bisection method[7], discussed in this section)
5. UVa 10474 - Where is the Marble? (very simple: use `sort` and then `lower_bound`)
6. UVa 10611 - Playboy Chimp (binary search)
7. UVa 10706 - Number Sequence (binary search + some mathematics insights)
8. UVa 10742 - New Rule in Euphomia (use sieve to generate prime numbers; binary search)
9. UVa 11057 - Exact Sum (sort, for price p[i], check if price (M-p[i]) exists with bsearch)
10. UVa 11413 - Fill the Containers (binary search the answer + simulation)
11. UVa 11876 - N + NOD (N) (use `[lower|upper]_bound` on sorted sequence N)
12. UVa 11881 - Internal Rate of Return (bisection method, there is only 1 possible answer)
13. UVa 11935 - Through the Desert (binary search the answer + simulation)
14. LA 2565 - Calling Extraterrestrial ... (Kanazawa02, binary search + math)
15. IOI 2010 - Quality of Living (binary search the answer)
16. IOI 2011 - Race (Divide & Conquer; solution path uses a certain vertex or not)
17. IOI 2011 - Valley (practice task; ternary search)
18. Thailand ICPC National Contest 2009 - My Ancestor (problem author: Felix Halim)

Also see: Data Structures with Divide & Conquer flavor (see Section 2.2.2)

Also see: Divide & Conquer for Geometry Problems (see Section 7.4)

---

[7]For alternative solutions for this problem, see `http://www.algorithmist.com/index.php/UVa_10341`.

3.4 Greedy

An algorithm is said to be greedy if it makes locally optimal choice at each step with the hope of eventually finding the optimal solution. For some cases, greedy works - the solution code becomes short and runs efficiently. But for *many* others, it does not. As discussed in typical CS textbooks [3, 23], a problem must exhibit these two ingredients 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 a greedy property (hard to prove its correctness!).

   If we make a choice that seems best at the moment and solve the remaining subproblems later, we still reach optimal solution. We never have to reconsider our previous choices.

3.4.1 Examples

Coin Change - Greedy Version

Suppose we have a large number of coins with different denominations, i.e. 25, 10, 5, and 1 cents. We want to make change using *least number of coins*. The following greedy algorithm works for this set of denominations of this problem: Keep using the largest denomination of coin which is not greater than the remaining amount to be made. Example: If the denominations are {25, 10, 5, 1} cents and we want to make a change of 42 cents, we can do: 42-25 = 17 → 17-10 = 7 → 7-5 = 2 → 2-1 = 1 → 1-1 = 0, of total 5 coins. This is optimal.

The coin changing example above has the two ingredients for a successful greedy algorithm:

1. It has optimal sub-structures.
   We have seen that in the original problem to make 42 cents, we have to use 25+10+5+1+1. This is an optimal 5 coins solution to the original problem!
   Now, the optimal solutions to its sub-problems are contained in this 5 coins solution, i.e.
   a. To make 17 cents, we have to use 10+5+1+1 (4 coins, part of solution for 42 cents above),
   b. To make 7 cents, we have to use 5+1+1 (3 coins, also part of solutions for 42 cents), etc

2. It has a greedy property.
   Given every amount V, we greedily subtract it with the largest denomination of coin 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.

However, this greedy algorithm does *not* work for *all* sets of coin denominations, e.g. {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 using 2 coins {3, 3}. This problem is revisited later in Section 3.5.2 (DP).

UVa 410 - Station Balance

Given $1 \leq C \leq 5$ chambers which can store 0, 1, or 2 specimens, $1 \leq S \leq 2C$ specimens, and M: a list of mass of the S specimens, determine in which chamber we should store each specimen in order to minimize IMBALANCE. See Figure 3.5 for visual explanation[8].

A = $(\sum_{j=1}^{S} M_j)/C$, i.e. A is the average of all mass over C chambers.

IMBALANCE = $\sum_{i=1}^{C} |X_i - A|$, i.e. sum of differences between the mass in each chamber w.r.t A where X_i is the total mass of specimens in chamber i.

This problem can be solved using a greedy algorithm. But to arrive at that solution, we have to make several observations.

[8]Since $C \leq 5$ and $S \leq 10$, then we can actually use a Complete Search solution for this problem. However, this problem is simpler to be solved using Greedy algorithm.

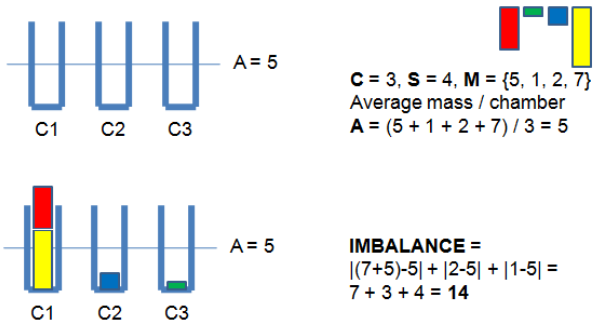


Figure 3.5: Visualization of UVa 410 - Station Balance

Observation: 1). If there exists an empty chamber, at least one chamber with 2 specimens must be moved into this empty chamber! Otherwise the empty chamber contributes too much to IMBALANCE! See Figure 3.6, top. 2). If $S > C$, then $S - C$ specimens must be paired with one other specimen already in some chambers. The Pigeonhole principle! See Figure 3.6, bottom.

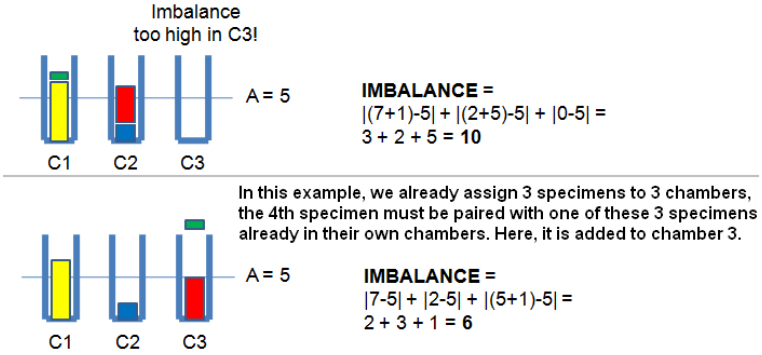


Figure 3.6: UVa 410 - Observation 1

Now, the key insight that can simplify the problem is this: If $S < 2C$, add dummy $2C-S$ 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 these specimens based on their mass such that $M_1 \leq M_2 \leq \ldots \leq M_{2C-1} \leq M_{2C}$. In this example, $M = \{5, 1, 2, 7, 0, 0\} \rightarrow \{0, 0, 1, 2, 5, 7\}$.

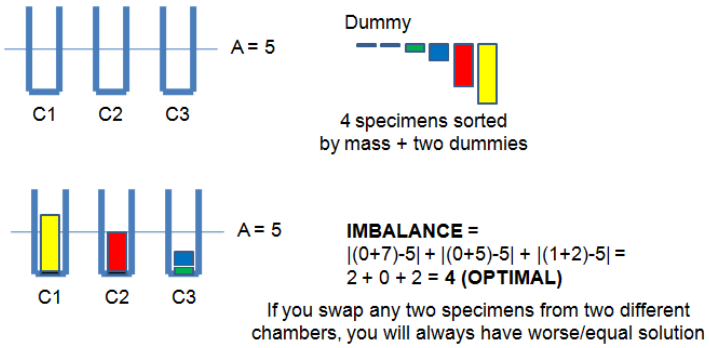


Figure 3.7: UVa 410 - Greedy Solution

By adding dummy specimens and then sorting them, a greedy strategy 'appears'. We can now:
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 3.7.
It is hard to teach the technique to come up with this greedy solution. Finding greedy solution can sometimes be an 'art'. One tip from this example: If no obvious greedy strategy is seen, try to *sort* the data first or introduce some tweaks and see if a greedy strategy emerges.

Exercise 3.4.1.1: Attempt this UVa 410 and its similar 'load balancing' variant: UVa 11389.

UVa 10382 - Watering Grass

Problem description: n sprinklers are installed in a horizontal strip of grass L meters long and W meters wide. Each sprinkler is installed at the horizontal center line of 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 to turn on in order to water the entire strip of grass? Constraint: $n \leq 10000$. For a test case of this problem, see Figure 3.8 (left). The answer for this test case is 6 sprinklers (those labeled with {A, B, D, E, F, H}). There are 2 unused sprinklers for this test case (those labeled with {C, G}).

We cannot solve this problem by using a brute force strategy that tries all possible subsets of sprinklers to be turned on. This is because the number of sprinklers can go up to 10000. It is impossible to try all 2^{10000} possible subsets of sprinklers to get the answer.

This problem is actually a well known greedy problem called the *interval covering* problem. However there is a geometric twist. The original interval covering problem deals with intervals. But this problem deals with sprinklers with circle radius of operation and a horizontal strip of grass. So we have to first transform the problem into the interval covering problem.

See Figure 3.8 (right). We can convert these circles and horizontal strip into intervals. We can compute `dx = sqrt(`R^2 `-` $(W/2)^2$`)`. Then, suppose a circle is centered at `(x, y)`, the interval represented by this circle is actually `[x-dx..x+dx]`. If you have difficulties with this geometric transformation, see Section 7.2.4 that discusses basic operations on *right triangle*.

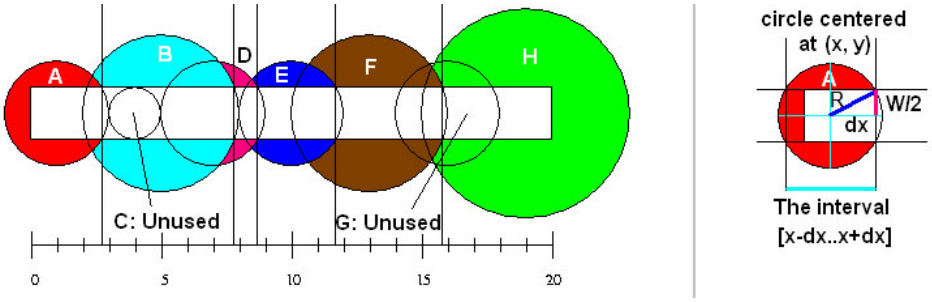


Figure 3.8: UVa 10382 - Watering Grass

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 based on *increasing* left endpoint, and if tie, by *decreasing* right endpoint. Then, the Greedy algorithm scans the interval one by one. It takes the interval that covers the 'as far right as possible' yet still form an uninterrupted coverage from the leftmost side to the rightmost side of the horizontal strip of grass. It ignores intervals that are already covered by other intervals.

For the test case shown in Figure 3.8 (left), this Greedy algorithm first sorts the intervals into {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 of 'A'), ignores 'C' (as it is embedded inside interval of 'B'), takes 'D' (it has to, as the interval of 'B' and 'E' are not connected if it does not take interval of 'D'), takes 'E', takes 'F', ignores 'G' (as taking interval of '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 of 'F' and covers more to the right than interval of 'G' and goes beyond the rightmost side of the grass strip). In total, we take 6 sprinklers: {A, B, D, E, F, H}. This is the minimum possible for this test case.

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 only), Load Balancing, and Interval Covering. There are two other classical examples of Greedy algorithms in this book, e.g. Kruskal's (and Prim's) algorithm for Minimum Spanning Tree (MST) problem (see Section 4.3) and Dijkstra's algorithm for Single-Source Shortest Paths (SSSP) problem (see Section 4.4.3). There are more known Greedy algorithms that we choose not to discuss in this book as they are too 'problem specific' and rarely appear in programming contests, e.g. Huffman Codes [3, 23], Fractional Knapsack [3, 23].

However, today's programming contests (both ICPC and IOI) rarely ask for the pure version of these classical problems. Using Greedy algorithms to attack a 'non classical' problem is usually risky. A Greedy algorithm normally will not encounter TLE response, as it is lightweight, but tends to get WA response. Proving that a certain 'non classical' problem has optimal sub-structure and greedy property during contest time may be time consuming, so a competitive programmer usually do this instead:

He will look at the input size. If it is 'small enough' for the time complexity of either Complete Search or Dynamic Programming algorithm (see Section 3.5), he will use one of these approaches as both will ensure a correct answer. He will *only* use a Greedy algorithm if he knows for sure that the input size given in the problem statement is too large for his best Complete Search or DP solution algorithm.

Having said that, it is quite true that many problem authors nowadays set the input size of such can-use-Greedy-algorithm-or-not-problems to be in some reasonable range so the contestants *cannot* use the input size to quickly determine the required algorithm!

Programming Exercises solvable using Greedy (hints omitted to keep the problems challenging):

1. UVa 00311 - Packets
2. UVa 00410 - Station Balance (discussed in this section)
3. UVa 10020 - Minimal Coverage
4. UVa 10026 - Shoemaker's Problem
5. UVa 10152 - ShellSort
6. UVa 10249 - The Grand Dinner
7. UVa 10340 - All in All
8. UVa 10382 - Watering Grass (discussed in this section)
9. UVa 10440 - Ferry Loading II
10. UVa 10602 - Editor Nottobad
11. **UVa 10656 - Maximum Sum (II) ***
12. UVa 10670 - Work Reduction
13. UVa 10672 - Marbles on a tree
14. UVa 10700 - Camel Trading
15. UVa 10714 - Ants
16. UVa 10718 - Bit Mask
17. UVa 10747 - Maximum Subsequence
18. UVa 10763 - Foreign Exchange
19. UVa 10785 - The Mad Numerologist
20. UVa 11054 - Wine Trading in Gergovia
21. UVa 11103 - WFF'N Proof
22. **UVa 11157 - Dynamic Frog ***
23. UVa 11292 - Dragon of Loowater
24. UVa 11369 - Shopaholic
25. **UVa 11389 - The Bus Driver Problem ***
26. UVa 11520 - Fill the Square
27. UVA 11532 - Simple Adjacency ...
28. UVa 11567 - Moliu Number Generator
29. UVa 11729 - Commando War
30. UVa 11900 - Boiled Eggs
31. LA 2519 - Radar Installation (Beijing02)
32. IOI 2011 - Elephants (optimized greedy solution[9] can be used up to subtask 3)

[9]But the harder subtasks 4 and 5 must be solved using efficient data structure.

3.5 Dynamic Programming

Dynamic Programming (from now on abbreviated as DP) is perhaps the most challenging problem solving paradigm among the four paradigms discussed in this chapter. Therefore, make sure that you have mastered the material mentioned in the previous chapters/sections before continuing. Plus, get yourself ready to see lots of recursions and recurrence relations!

If you are new with DP technique, you can start by assuming that DP is a kind of 'intelligent' and 'faster' recursive backtracking. DP is primarily used to solve *optimization* problem and *counting* problem. So if you encounter a problem that says "minimize this" or "maximize that" or "count how many ways", then there is a chance that it is a DP problem. We will refine our understanding of Dynamic Programming throughout this section.

3.5.1 DP Illustration

We illustrate the concept of Dynamic Programming by using an example problem: UVa 11450 - Wedding Shopping. The abridged problem statement: Given different models for each garment (e.g. 3 shirt models, 2 belt models, 4 shoe models, ...) and a certain *limited* budget, *buy one model of each garment*. We cannot spend more money than the given budget, but we want to spend *the maximum possible*.

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. Then, information is given about the C garments. For a garment `g` $\in$ `[0..`C`-1]`, we know an integer $1 \leq K \leq 20$ which indicates the number of different models 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 to buy one element of each garment *without exceeding the budget*. If there is no solution due to that small amount of budget given to us, then just print "`no solution`".

For example, if the input is like this (test case A):
$M = 20$, $C = 3$
Price of the 3 models of garment `g = 0` → 6 4 <u>8</u> // see that 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

Then the answer is 19, which *may* come from buying the <u>underlined</u> items (8+10+1).
Note that this solution is not unique, as solution (6+10+3) and (4+10+5) are also optimal.

However, if the input is like this (test case B):
$M =$ **9 (very 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

Then the answer is "`no solution`" because even if we buy all the cheapest models for each garment, the total prices (4+5+1) = 10 is still larger than the given budget $M = 9$.

Exercise 3.5.1.1: To verify your understanding, what is the output for this test case C below?

Test case C:
$M = 25$, $C = 3$
Price of the 3 models of garment `g = 0` → 6 4 8
Price of the 2 models of garment `g = 1` → 10 6
Price of the 4 models of garment `g = 2` → 7 3 1 5

In order for us to appreciate the usefulness of Dynamic Programming for solving the above-mentioned problem, let us first explore how the *other* paradigms fare in this particular problem.

Approach 1: Complete Search (Time Limit Exceeded)

First, let's see if Complete Search (recursive backtracking) can solve this problem. One way to write recursive backtracking for 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.

We start with `money = M` and garment `g = 0` and try all possible models in garment `g = 0` one by one (maximum 20 models). If model i is chosen, we subtract `money` with model i's price, and then recursively do the same process to garment `g = 1` (which also can go up to 20 models), etc. We stop if the model for the last garment `g = C-1` has been chosen. If `money < 0` before we reach the last garment `g = C-1`, prune this infeasible solution. Among all valid combinations, pick one that makes `money` as close to 0 as possible yet still non negative. This maximizes the money spent, which is (`M - money`).

We can formally define these Complete Search recurrences (transitions) as follow:

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 `g = C-1` has been bought (i.e. a candidate solution),
`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 **very slow**! Let's analyze its 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 the last garment `g = 19` ***also*** has up to 20 models. Therefore, Complete Search like this runs in $20 \times 20 \times \ldots \times 20$ of total 20 times in the worst case, i.e. 20^{20} = a **very very large** number. If we can *only* come up with this Complete Search solution, we cannot solve this problem.

Approach 2: 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 in each garment `g` which still fits our budget. For example in test case A above, we 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 `money` left does not allow us to buy 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, which is $20 + 20 + \ldots + 20$ of total 20 times = 400 operations in the worst test case. However, this greedy strategy does not work for many other test cases, like this counter-example below (test case D):

Test case D:
$M = 12$, $C = 3$
3 models of garment `g = 0` $\rightarrow$ 6 $\underline{4}$ **8**
2 models of garment `g = 1` $\rightarrow$ $\underline{5}$ 10
4 models of garment `g = 2` $\rightarrow$ 1 5 $\underline{3}$ 5

Greedy strategy selects model 3 of garment `g = 0` with price **8** (`money = 12-8 = 4`), thus we do not have enough money to buy any model in garment `g = 1` and wrongly reports "`no solution`". The optimal solution is actually ($\underline{4+5+3} = 12$), which uses all our budget.

Approach 3: Top-Down DP (Accepted)

To solve this problem, we have to use DP. We start by discussing the Top-Down[10] version.

[10]In simple, you can understand 'top-down' DP as recursive backtracking that is more 'intelligent' and 'faster' as it does not recompute overlapping sub-problems.

This problem has the two ingredients to make DP works:

1. This problem has optimal sub-structures[11].
   This is shown in the Complete Search recurrence 3 above: The solution for the sub-problem is part of the solution of the original problem.
2. This problem has overlapping sub-problems.
   This is the key point of DP! The search space of this problem is *not* as big as the 20^{20} operations that we have estimated in our Complete Search discussion above. This is because **many** sub-problems are *overlapping*!

Let's verify if this problem indeed has overlapping sub-problems. Suppose that there are 2 models in certain garment `g` with the *same* price p. Then, Complete Search will move to the **same** sub-problem `shop(money - p, g + 1)` after picking *either* model! Similarly, this situation also occur if some combination of `money` and chosen model's price causes $\texttt{money}_1 - \texttt{p}_1 = \texttt{money}_2 - \texttt{p}_2$. This will cause the same sub-problems to be computed more than once! This is inefficient!

So, how many *distinct* sub-problems (a.k.a. **states** in DP terminology) are there in this problem? The answer is only $201 \times 20 = 4020$. There are only 201 possible values for `money` (from 0 to 200, inclusive) and 20 possible values for garment `g` (from 0 to 19, inclusive). Each sub-problem just need 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 Approach 1 above), we can implement the **top-down** DP by adding these few additional steps:

1. Initialize[12] a DP 'memo' table with dummy values that are not used in the problem, e.g. '-1'. The dimension of the DP table must be the size of distinct sub-problems (states).
2. At the start of recursive function, simply check if this current state has been computed before.
   (a) If it is, simply return the value from the DP memo table, $O(1)$.
   Hence the other name of top-down DP is 'memoization' technique.
   (b) If it is not, compute as per normal (just once) and then store the computed value in the DP memo table so that further calls to this sub-problem (state) is fast.

Analyzing a basic[13] DP solution is easy. If it has M distinct states, then it requires $O(M)$ memory space. If computing one state requires $O(k)$ steps, then the overall time complexity is $O(kM)$. The UVa 11450 - Wedding Shopping problem above 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 $4020 \times 20 = 80400$ operations, which is very manageable.

We show 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 with the recursive backtracking code that you have learned in Section 3.2.

```
/* UVa 11450 - Wedding Shopping - Top Down */
// this code is similar to recursive backtracking code
// parts of the code specific to top-down DP are commented with word: 'TOP-DOWN'
#include <algorithm>
#include <cstdio>
#include <cstring>
using namespace std;

int M, C, price[25][25];                        // price[g (<= 20)][model (<= 20)]
int memo[210][25];          // TOP-DOWN: dp table memo[money (<= 200)][g (<= 20)]
```

[11] Optimal sub-structures is the same ingredient needed to make a Greedy algorithm work, but this problem lacks the 'greedy property' ingredient which makes this problem unsuitable for Greedy algorithm.

[12] For C++ users, `memset` function is a good way to perform this step.

[13] Basic means: "Without fancy optimizations that we will see later in this section and in Section 8.4".

```
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 (memo[money][g] != -1)               // TOP-DOWN: this state has been visited
    return memo[money][g]; // TOP-DOWN: simply return the value in DP memo table
  int ans = -1000000000;
  for (int model = 1; model <= price[g][0]; model++)  // try all possible models
    ans = max(ans, shop(money - price[g][model], g + 1));
  return memo[money][g] = ans;  // TOP-DOWN: assign ans to dp table + 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]);   // to simplify coding, 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 (~ recursive backtracking++)
    if (score < 0) printf("no solution\n");
    else           printf("%d\n", score);
} } // return 0;
```

Example codes: ch3_03_UVa11450_td.cpp; ch3_03_UVa11450_td.java

Approach 4: Bottom-Up DP (Accepted)

There is another style of writing DP solutions, called the **bottom-up** DP.

This is actually the 'true form' of DP as DP is originally known as 'tabular method' (computation technique involving a table). The steps to build bottom-up DP solution are:

1. Determine the required set of parameters that uniquely describe sub-problem. If there are N parameters, then prepare an N dimensional DP table, one cell per state.
2. Initialize some cells of the DP table with known initial values (the base cases).
3. With some cells/states in the DP table already filled, determine which other cells/states can be filled next. This process is repeated until the DP table is complete.

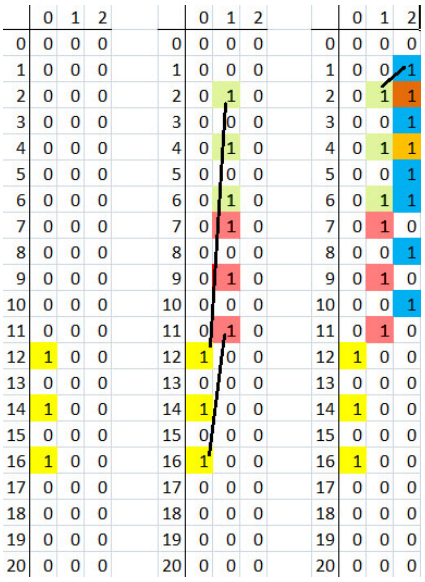


Figure 3.9: Bottom-Up DP

For UVa 11450, we can write bottom-up DP as follow: We describe the state of a sub-problem by using two parameters: Current `money` and garment `g`, same as top-down DP above. Then, we set a 2D table (boolean matrix) `can_reach[money][g]` of size 201 × 20. Initially, only the cells/states reachable by buying any of the models of the first garment `g = 0` are set to true. Let's use test case A above as example. In Figure 3.9 leftmost, only rows '20-6 = 14', '20-4 = 16', and '20-8 = 12' in column 0, are initially set to true.

Now, we loop through from the second garment `g = 1` until the last garment `g = C-1`. We set `can_reach[a][b]` to be true if it is possible to reach this state from any states in the previous column, i.e. from state `can_reach[a +` price of any model of current garment `g][b - 1]`. See Figure 3.9, middle, where for example, `can_reach[11][1]` can be reached from `can_reach[11 + 5][0]` by buying a model with price 5 in garment `g = 1`; `can_reach[2][1]` can be reached from `can_reach[2 + 10][0]` by buying a model with price 10 in garment `g = 1`; etc. We repeat this table filling process column by column until we are done with the last column[14].

Finally, the answer can be found in the last column `C-1`. Find the cell in that column that is nearest to index 0 that is set to true. In Figure 3.9, rightmost, the cell `can_reach[1][2]` is the answer. This means that we can somehow reach this state (`money = 1`) by buying some combination of various garment models. The final answer is actually `M - money`, or in this case, 20-1 = 19. The answer is "`no solution`" if there is no cell in the last column that is set to be true.

We provide our code in below for comparison with the top-down version.

```
/* UVa 11450 - Wedding Shopping - Bottom Up */
#include <cstdio>
#include <cstring>
using namespace std;

int main() {
  int i, j, l, TC, M, C, price[25][25];       // price[g (<= 20)][model (<= 20)]
  bool can_reach[210][25];         // can_reach table[money (<= 200)][g (<= 20)]

  scanf("%d", &TC);
  while (TC--) {
    scanf("%d %d", &M, &C);
    for (i = 0; i < C; i++) {
      scanf("%d", &price[i][0]);   // to simplify coding, store K in price[i][0]
      for (j = 1; j <= price[i][0]; j++) scanf("%d", &price[i][j]);
    }

    memset(can_reach, false, sizeof can_reach);              // clear everything
    for (i = 1; i <= price[0][0]; i++)                         // initial values
      can_reach[M - price[0][i]][0] = true; // if only using first garment g = 0

    // time complexity: O(C*M*K) = 20*201*20 = 80400 => same as top-down version
    for (j = 1; j < C; j++)   // for each remaining garment (note: column major)
      for (i = 0; i < M; i++) if (can_reach[i][j - 1])   // can reach this state
        for (l = 1; l <= price[j][0]; l++) if (i - price[j][l] >= 0) // feasible
          can_reach[i - price[j][l]][j] = true; // flag this state as reachable

    for (i = 0; i <= M && !can_reach[i][C - 1]; i++);  // ans is in the last col

    if (i == M + 1) printf("no solution\n");    // last col has no bit turned on
    else            printf("%d\n", M - i);
} } // return 0;
```

Example codes: `ch3_04_UVa11450_bu.cpp`; `ch3_04_UVa11450_bu.java`

Exercise 3.5.1.2: Rewrite the main loop of the DP bottom up code shown in this section from column major to row major! See Tips 6.3 (Optimizing Your Source Code) in Section 3.2.2.

[14]Later in Section 4.7.1, we will see DP as a traversal on (implicit) DAG. To avoid unnecessary 'backtrack' along this DAG, we have to visit the vertices according to its topological order (see Section 4.2.5). The order in which we fill the DP table is the topological order of the underlying implicit DAG.

Approach 5: Bottom-Up DP - Space Saving Trick (Accepted)

There is an advantage of writing DP solution in bottom-up fashion. For some problems where we only need the best solution value – including this UVa 11450, we can optimize the memory usage of our DP solution by sacrificing one DP dimension. For some harder DP problems which have high memory requirement, this 'space saving trick' may be useful.

Let's take a look at Figure 3.9. To compute column 1, we only need to know which rows in column 0 are set to true. To compute column 2, we only need to know which rows in column 1 are set to true. In general, to compute column i, we only need the values in the previous column $i-1$. So, instead of storing a boolean matrix `can_reach[money][g]` of size 201×20, we just need to store `can_reach[money][2]` of size 201×2. We can use a programming trick to set one column as 'previous' and the other column as 'current' (e.g. `prev = 0, cur = 1`) and then swap them (e.g. now `prev = 1, cur = 0`) as we compute the bottom-up DP column by column.

Exercise 3.5.1.3: After completing **Exercise 3.5.1.2** above (thus `can_reach[money][g]` becomes `can_reach[g][money]`), use only *two* rows instead of twenty and solve UVa 11450.

Top-Down versus Bottom-Up DP

As you can see, although both styles uses 'table', the way bottom-up DP table is filled is different compared to top-down DP *memo* table. In top-down DP, the memo table entries are filled 'as needed' and this is mainly done by the recursion itself. In bottom-up DP, we have to come up with the correct 'DP table filling order', usually by sequencing the (nested) loops properly.

For most DP problems, these two styles are equally good and the decision on using which DP style is in your hand. For some harder DP problems, one of the style is better over another. To help you decide which style that you should take when presented with a DP problem, we present the trade-off comparison between top-down and bottom-up DP 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 recursive calls overhead (usually this is not penalized in programming contests) 2. If there are M states, it can use up to $O(M)$ table size which can lead to Memory Limit Exceeded (MLE) for some harder problems	Cons: 1. For some 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

Reconstructing the Optimal Solution

In many DP problems, usually only the optimal solution value is asked. However, many contestants are caught off-guard when they are also required to print the optimal solution too. To do this, we must memorize the predecessor information at each state. Then, once we have the optimal final state, we do backtracking from that final state, following the optimal transition recorded at each state, until we reach the starting state again.

Example: See Figure 3.9, right. We know that the optimal final state is `can_reach[1][2]`. We remember that this state is reachable from `can_reach[2][1]`, so we backtrack to `can_reach[2][1]`. Now, see Figure 3.9, middle. State `can_reach[2][1]` is reachable from `can_reach[12][0]`, so we backtrack to `can_reach[12][0]`. As this is already one of the starting state (at first column), then we know the optimal solution is (20→12)=price 8, then (12→2)=price 10, then (2→1)=price 1.

3.5.2 Classical Examples

The problem UVa 11450 - Wedding Shopping above is a non classical DP problem where we have 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 must be mastered by every contestants who wish to do well in ICPC or IOI! In this section, we list down five classical DP problems and their solutions. Note: Once you understand the basic form of these DP solutions, try solving the programming exercises that discusses their *variants*.

1. Longest Increasing Subsequence (LIS)

Problem: Given a sequence {`X[0], X[1], ..., X[n-1]`}, determine its Longest Increasing Subsequence (LIS)[15] – as the name implies. Take note that 'subsequence' is not necessarily contiguous.

Example: $n = 8$, $X = \{-7, 10, 9, 2, 3, 8, 8, 1\}$
The LIS is {-7, 2, 3, 8} of length 4.

Index	0	1	2	3	4	5	6	7
X	-7	10	9	2	3	8	8	1
LIS(i)	1	2	2	2	3	4	4	2

Figure 3.10: Longest Increasing Subsequence

One solution: Let `LIS(i)` be the LIS ending in index `i`, then we have these recurrences:
1. `LIS(0) = 1` // base case
2. `LIS(i) = 1 + max(LIS(j))`, $\forall$ `j` $\in$ `[0..i-1]` and `X[j]` $<$ `X[i]` // recursive case

The answer is the highest value of `LIS(k)` $\forall$ `k` $\in$ `[0..n-1]`.

In Figure 3.10, we see that:

- `LIS(0)` is 1, the number {-7} itself.
- `LIS(1)` is now 2, as we can extend {-7} with {10} to form {-7, 10} of length 2.
- `LIS(2)` is also 2, as we can stay with {-7, 10} or form {-7, 9}.
  We cannot form {-7, 10} + {9} as it is non increasing.
- `LIS(3)` is also 2, as we can stay with {-7, 10} or {-7, 9}; or form {-7, 2}.
  We cannot form {-7, 10} + {2} or {-7, 9} + {2} as both are non increasing.
- `LIS(4)` is now 3, as we can extend {-7, 2} with {3} to form {-7, 2, 3}.
  This is the longest among other possibilities.
- `LIS(5)` is now 4, as we can extend {-7, 2, 3} + {8} to form {-7, 2, 3, 8}.
  This is the longest among other possibilities.
- `LIS(6)` is also 4, as we can stay with {-7, 2, 3, 8}.
  We cannot form {-7, 2, 3, 8} + {8} as the last two items are the same.
- `LIS(7)` is also 4, as we can stay with {-7, 2, 3, 8}.
  We cannot form {-7, 2, 3, 8} + {1} as it is non increasing.
- The answers are in `LIS(5)` or `LIS(6)`, both with value (LIS length) = 4.

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, i.e. the LIS ending at index `i`, $\forall$ `i` $\in$ `[0..n-1]`. As we need to compute each state with an $O(n)$ loop, then 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 3.10) and then trace back the arrows from index `k` that contains the highest value of `LIS(k)`. For example, `LIS(5)` is the optimal final state. Check Figure 3.10. 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}.

[15]There are other variants of this problem: Longest *Decreasing* Subsequence, Longest *Non Increasing/Decreasing* Subsequence. Note that increasing subsequences can be modeled as a Directed Acyclic Graph (DAG). Thus finding LIS is equivalent to finding the Longest Paths in DAG (see Section 4.7.1).

The LIS problem can be solved in *output-sensitive* $O(n \log k)$ (where k is the length of the LIS) instead of $O(n^2)$ by realizing that the LIS is sorted and we can use binary search to find the appropriate insertion point to extend *or update* the LIS. Let array `A` contains the current LIS so far. Using the same example as above, we will update array `A` step by step as follow:

- Initially, at `X[0] = -7`, we have `A = {-7}`
- We can insert `X[1] = 10` at `A[1]` so that we have a longer LIS `A = {-7, 10}`.
- For `X[2] = 9`, we replace `A[1]` so that we have a 'better' LIS `A = {-7, 9}`.
  This is a *greedy* strategy. By having an LIS with smaller ending value, we maximizes our chance to further extend the LIS with other future values later.
- For `X[3] = 2`, we replace `A[1]` again so that we have an 'even better' LIS `A = {-7, 2}`.
- We can insert `X[4] = 3` at `A[2]` so that we have a longer LIS `A = {-7, 2, 3}`.
- We can insert `X[5] = 8` at `A[3]` so that we have a longer LIS `A = {-7, 2, 3, 8}`.
- For `X[6] = 8`, nothing change as `A[3] = 8`. Our LIS is still `A = {-7, 2, 3, 8}`.
- For `X[7] = 1`, we actually change `A[1]` so that `A = {-7, 1, 3, 8}`.
  This array `A` is no longer the LIS of `X`, but this step is important as there can be longer subsequence in the future that may start with `A[1] = 1`. For example, try this test case, `X = {-7, 10, 9, 2, 3, 8, 8, 1, 2, 3, 4}`. The length of LIS for this test case is 5.
- The answer is the largest size of array `A` throughout this process.

Example codes (Longest *Non*-Increasing Subsequence): `ch3_05_UVa231.cpp`; `ch3_05_UVa231.java`

2. Max Sum

Abridged problem statement of UVa 108 - Maximum Sum: Given an $n \times n$ ($1 \leq n \leq 100$) square matrix of integers, each ranging from `[-127..127]`, find a sub-matrix with the maximum sum.

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 15 (9 + 2 - 4 + 1 - 1 + 8). There is no other sub-matrix with larger sum.

A					B					C				
	0	-2	-7	0		0	-2	-9	-9		0	-2	-9	-9
	9	2	-6	2		9	9	-4	2		9	9	-4	2
	-4	1	-4	1		5	6	-11	-8		5	6	-11	-8
	-1	8	0	-2		4	13	-4	-3		4	13	-4	-3

Table 3.3: UVa 108 - Maximum Sum

Attacking this problem naïvely using iterative brute force as shown below does not work as it needs $O(n^6)$. For the largest test case with $n = 100$, $O(n^6)$ 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 coordinate
    subRect = 0;                                  // sum items in this sub-rectangle
    for (int a = i; a <= k; a++) for (int b = j; b <= l; b++)
      subRect += arr[a][b];
    maxSubRect = max(maxSubRect, subRect); }                // the answer is here
```

Solution: There is a well-known DP solution for a *static* range problem like this. DP can work for this problem as computing a large sub-matrix will definitely involves computing smaller sub-matrices in it and such computation involves *overlapping* sub-matrices!

One possible DP solution is to turn this $n \times n$ matrix into an $n \times n$ *sum matrix* where `arr[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 easily done simultaneously when reading the input and still runs in $O(n^2)$. The code shown below turns input square matrix (see Table 3.3.A) into 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", &arr[i][j]);
  if (i > 0) arr[i][j] += arr[i - 1][j];     // if possible, add values from top
  if (j > 0) arr[i][j] += arr[i][j - 1];    // if possible, add values from left
  if (i > 0 && j > 0) arr[i][j] -= arr[i - 1][j - 1];   // to avoid double count
}                                                       // inclusion-exclusion principle
```

Now, with this sum matrix, we can answer the sum of any sub-matrix `(i, j)` to `(k, l)` in $O(1)$. Suppose we want to know the sum of `(1, 2)` to `(3, 3)`. We split the sum matrix into 4 sections and compute `arr[3][3] - arr[0][3] - arr[3][1] + arr[0][1] = -3 - 13 - (-9) + (-2) = -9` as highlighted in Table 3.3.C. With this $O(1)$ DP formulation, this problem can now be solved in $O(n^4)$. For the largest test case with $n = 100$, this is 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 coordinate
    subRect = arr[k][l]; // this is sum of all items from (0, 0) to (k, l): O(1)
    if (i > 0) subRect -= arr[i - 1][l];                                  // O(1)
    if (j > 0) subRect -= arr[k][j - 1];                                  // O(1)
    if (i > 0 && j > 0) subRect += arr[i - 1][j - 1];                     // O(1)
    maxSubRect = max(maxSubRect, subRect); }               // the answer is here
```

Example codes: ch3_06_UVa108.cpp; ch3_06_UVa108.java

Not every range problems require Segment or Fenwick Tree as in Section 2.3.3 or 2.3.4! Problems where the input data is static like this is usually solvable with DP technique. It is also worth mentioning that the solution for this problem is more natural to be written with bottom-up DP technique as we are already working with matrix and loops in the first place. We can write recursive top-down solution for this problem, but it will not look natural.

Exercise 3.5.2.1: The solution above runs in $O(n^4)$. Actually, there exists an $O(n^3)$ solution that uses a certain *greedy property* of this problem. Can you figure out how to do it?

Exercise 3.5.2.2: What if the given static array is 1D? Can you form a similar $O(1)$ DP solution to answer `Range Sum Query(i, j)`, i.e. `RSQ(i, j) = arr[i] + arr[i+1] + ...+ arr[j]`?

Exercise 3.5.2.3: Use the solution from **Exercise 3.5.2.2** to find the maximum sum in 1D array in $O(n^2)$. More challenging: Can you make the solution runs in $O(n)$ by using the similar greedy property used earlier in **Exercise 3.5.2.1**?

Exercise 3.5.2.4: The solution for `Range Minimum Query(i, j)` on 1D array in Section 2.3.3 uses Segment Tree. Utilize DP technique to answer `RMQ(i, j)`. Assume that the data is static!

3. 0-1 Knapsack (Subset Sum)

Problem[16]: 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 one can carry, if he can either[17] ignore or take a particular item (hence the term 0-1 for ignore/take).

[16]This problem is also known as the Subset Sum problem. It has similar problem description: Given a set of integers and an integer S, does any non-empty subset sum to S?

[17]There are other variants of this problem, e.g. the Fractional Knapsack problem solvable with Greedy algorithm.

Example: $N = 3$, $V = \{100, 70, 50\}$, $W = \{10, 5, 7\}$, $S = 12$.
If we select item 0 with weight 10 and value 100, we cannot take any other item. We get 100.
If we select item 1 and 2, we have total weight 12 and total value 120. This is the maximum.

Solution: Use these Complete Search recurrences `value(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. `value(id, 0) = 0` // if `remW = 0`, we cannot take anything else
2. `value(N, remW) = 0` // if `id = N`, we have considered all items; no more item to deal with
3. if `W[id]` $>$ `remW`, we have no choice but to ignore this item
`value(id, remW) = value(id + 1, remW)`
4. if `W[id]` $\leq$ `remW`, we have two choices: ignore or take this item; we take the maximum
`value(id, remW) = max(value(id + 1, remW), V[id] + value(id + 1, remW - W[id]))`

Again, there are many overlapping sub-problems in this 0-1 Knapsack problem, but there are only $O(NS)$ possible distinct states (as `id` can vary between `[0..N-1]` and `curW` can vary between `[0..S]`)! We can compute each state in $O(1)$, thus the overall time complexity of this DP solution is $O(NS)$[18]. The answer is in `value(0, S)`.

Note: The top-down version of this DP solution is faster than the bottom-up version. This is because not all states are actually visited. Remember: Top-down DP only visits *the required states* whereas bottom-up DP visits *all distinct states*. Both versions are shown in our example codes.

Example codes: `ch3_07_UVa10130.cpp`; `ch3_07_UVa10130.java`

4. Coin Change (CC) - The General Version

Problem: Given a target amount V cents and a list of denominations of N coins, i.e. We have `coinValue[i]` (in cents) for coin type `i` $\in$ `[0..N-1]`, what is the minimum number of coins that we must use to obtain amount V? Assume that we have unlimited supply of coins of any type.

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 Greedy algorithm if the coin denominations are suitable (see Section 3.4). Example 1 is solvable with Greedy algorithm. But for general cases, we have to use DP. See Example 2 below:

Example 2: $V = 7$, $N = 4$, `coinValue` $= \{1, 3, 4, 5\}$
Greedy solution will answer 3, using 5+1+1 = 7, but optimal solution is 2, using 4+3 only!

Solution: Use these Complete Search recurrences `change(value)` where `value` is the current remaining cents that we want to change:

1. `change(0) = 0` // we need 0 coin to produce 0 cent
2. `change(`< 0`) =` $\infty \rightarrow$ (in practice, we just return a large positive value)
3. `change(value) = 1 + min(change(value - coinValue[i]))`; $\forall$ `i` $\in$ `[0..N-1]` // try all

The answer is in `change(V)`.
In Figure 4.2.3, we see that:
`change(0) = 0` and `change(`< 0`) =` ∞: base cases.
`change(1) = 1`, from `1 + change(1-1)`,
as `1 + change(1-5)` is infeasible.
`change(2) = 2`, from `1 + change(2-1)`,
as `1 + change(2-5)` is also infeasible.
... 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(10)`, which is 2.

<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 3.11: Coin Change

[18]Note that if S is large such that $NS >> 1M$, this DP solution is not feasible even with space saving trick!

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 ways* to get value V cents using a list of denominations of N coins. For Example 1 above, the answer is 3 ways: {ten 1 cent coins, one 5 cents coin plus five 1 cent coins, two 5 cents coins}.

Solution: Use these Complete Search recurrences `ways(type, value)` where `value` is the same as above but now we need one more parameter `type` to differentiate in which coin type that we are currently on. This second parameter is important because in this variant, we have to consider each coin type one after another. Once we choose to ignore a certain coin type, we should not reconsider it again:

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, coin type must be $\in$ `[0..N-1]`
4. `ways(type, value) = ways(type + 1, value) + ways(type, value - coinValue[type])` // general case: count the number of ways if we ignore this coin type plus if we use it

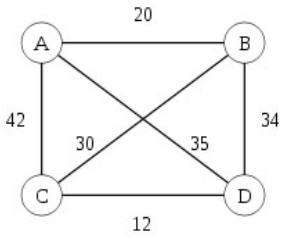
There are only $O(NV)$ possible distinct states. Each state can be computed in $O(1)$, thus the overall time complexity of this DP solution is $O(NV)$[19]. The answer is in `ways(0, V)`.

Example codes (this coin change variant): `ch3_08_UVa674.cpp`; `ch3_08_UVa674.java`

---

5. Traveling Salesman Problem (TSP)

Problem: Given N cities and their all-pairs distances in form of a matrix `dist` of size $N \times N$, compute the cost of making a tour[20] that starts from any city s, goes through the other $N-1$ cities *exactly once*, and finally return to the starting city s.



	A	B	C	D
A	0	20	42	35
B	20	0	30	34
C	42	30	0	12
D	35	34	12	0

Figure 3.12: TSP

Example: The TSP instance shown in Figure 3.12 has $N = 4$. Therefore, we have $4! = 24$ possible tour permutations. One of the minimum tour is `A-B-C-D-A` with cost `20+30+12+35 = 97` (notice that there can be more than one optimal solution).

Brute force TSP solution (either iterative or recursive) that tries all $O(n!)$ possible tours is only effective for $N = 10$ or at most $N = 11$ as $10! \approx 3M$ and $11! \approx 40M$. When $N > 11$, such solution will get TLE in programming contests.

We can utilize DP for TSP as the computation of sub-tours is clearly overlapping, e.g. one tour can be $A - B - C -$(*$N-3$ other cities that finally goes back to A*) and then another tour has $A - C - B -$*the same* ($N-3$) *other cities that also goes back 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 requires two parameters: The current city/vertex `pos` and something that we have not seen before: A *subset* of visited cities.

There are several ways to represent a set. But since we are going to pass this set information around as a parameter of a recursive function (for top-down DP), then this representation must be lightweight! In Section 2.2.1, we have been presented with such data structure: The *lightweight set of Boolean* (a.k.a `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 (has been visited). Vice versa if bit i is '0' (off).

[19]Note that if V is large such that $NV >> 1M$, this DP solution is not feasible even with space saving trick!

[20]Such tour is called the Hamiltonian tour: A cycle in an undirected graph which visits each vertex exactly once and also returns to the starting vertex.

For example: `bitmask`= $18_{10} = 10010_2$ implies that item (city) {1, 4} are in[21] the set (have been visited). Recall: To check if bit i in is on or off, we can use bitwise manipulation operator `bitmask & (1 << i)`. To set bit i, we can use `bitmask |= (1 << i)`.

Solution: Use these Complete Search recurrences `tsp(pos, bitmask)`:

1. `tsp(pos, `$2^N - 1$`) = dist[pos][0]` // all cities have been visited, return to starting city
// Note: `bitmask = `2^N`-1` implies that all N bits in `bitmask` are on.
2. `tsp(pos, bitmask) = min(dist[pos][nxt] + tsp(nxt, bitmask | (1 << nxt)))`
// $\forall$ `nxt` $\in$ `[0..N-1]`, `nxt != pos`, and `(bitmask & (1 << nxt))` is '0' (turned off)
// This recurrence basically tries all possible next cities that have not been visited before.

There are $O(N \times 2^N)$ distinct states and each state can be computed in $O(N)$, thus the overall time complexity of this DP solution is $O(N^2 \times 2^N)$. This allows us to solve up to[22] $N \approx 16$ 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 $11 \leq N \leq 16$, then DP is the solution, not brute force. The answer is in `tsp(0, 1)`: We start from city 0 (we can start from any vertex; the simplest choice is vertex 0) and set `bitmask = 1` so that city 0 is not re-visited again.

Usually, DP TSP problems require some kind of graph preprocessing to generate the distance matrix `dist` before running the DP solution. These variants are discussed in Section 8.2.2.

DP solution that involves a (small) set of Boolean as one of the parameter of its state is more well known as DP + bitmask technique. More challenging problems involving this technique are discussed in Section 8.4.

Example codes (solvable with DP TSP): `ch3_09_UVa10496.cpp`; `ch3_09_UVa10496.java`

3.5.3 Non Classical Examples

Although DP is one problem type that is most frequently appear in recent programming contests, the classical DP problems on their *pure forms* above usually never appear again in recent ICPCs or IOIs. We have to study them to understand more about DP. But once we do, we have to move on to solve many other non classical DP problems (which may become classic in the near future) and build up our 'DP technique skills'. In this subsection, we discuss two more non classical examples on top of the UVa 11450 - Wedding Shopping that has been discussed in detail earlier. We have also selected some easier non classical DP problems as programming exercises. Once you have cleared most problems, you are welcome to explore the more challenging ones in Section 8.4.

1. Cutting Sticks (UVa 10003)

Abridged problem statement: Given a stick of length $1 \leq L \leq 1000$, make $1 \leq n \leq 50$ cuts to that sticks (cut coordinates within range `[0..`L`]` are given). The cost of cut is determined by the length of the stick to be cut. Find a cutting sequence so that the overall cost is minimized!

Example: $L = 100$, $n = 3$, and cut coordinates: `arr` $= \{25, 50, 75\}$ (already sorted)

If we cut from left to right, then we will incur cost = 225
1. First, cut at coordinate 25, total cost = 100;
2. Then, cut at coordinate 50, total cost = 100 + 75 = 175;
3. Finally, cut at coordinate 75, total cost = 175 + 50 = 225;

However, the optimal answer is: 200
1. First, cut at coordinate 50, total cost = 100;
2. Then, cut at coordinate 25, total cost = 100 + 50 = 150;
3. Finally, cut at coordinate 75, total cost = 150 + 50 = 200;

[21] Remember that in `bitmask`, index starts from 0 and counted from the right.

[22] As programming contest problems require exact solutions, then the DP-TSP solution presented here is already one of the best solution. In real life, TSP instances go up to thousands cities. To solve large instances like that, we need non-exact approaches like the ones presented in [15].

Solution: First, append two more coordinates so that `arr` = {0, the original `arr`, and L}. Then, use these Complete Search recurrences `cut(left, right)` where `left`/`right` are the left/right indices of the current stick w.r.t `arr`, respectively. 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, ∀ i ∈ [0..n-1]` // If `left+1 = right` where `left` and `right` are the indices in `arr`, then we are left with one stick segment that do not need to be cut anymore.

2. `cut(left, right) = min(cut(left, i) + cut(i, right) + (arr[right] - arr[left]))` `∀ i ∈ [left+1..right-1]`
This recurrence basically tries all possible cutting points and pick the minimum one.
The cost of cut is the length of the current stick that is captured by `(arr[right] - arr[left])`.

This problem has overlapping sub-problems, but there are only $n \times n$ possible left/right indices or $O(n^2)$ distinct states. The cost to compute one state is $O(n)$. Thus, the overall time complexity is $O(n^3)$. As $n \leq 50$, this is feasible. The answer is in `cut(0, n+1)`.

Example codes: `ch3_10_UVa10003.cpp`; `ch3_10_UVa10003.java`

2. How do you add? (UVa 10943)

Abridged problem description: Given a number N, how many ways can K numbers 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 to add is ${}^{(n+k-1)}C_{(k-1)}$ (see Section 5.4.2 about Binomial Coefficients). Here, we will use this simple problem to re-illustrate Dynamic Programming principles that we have learned in this section.

First, we have to determine the parameters of this problem that have to be selected to represent distinct states of this problem. There are only two parameters in this problem, N and K and there are only 4 possible combinations:

1. If we do not choose any of them, we cannot represent distinct states. This option is ignored.
2. If we choose only N, then we do not know how many numbers $\leq N$ that have been used.
3. If we choose only K, then we do not know what is the target sum N.
4. Therefore, the distinct state of this problem is represented by a pair (N, K).

Next, we have to determine the base case(s) of this problem. 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. $N = X + (N - X)$. After doing this, we have 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 $N - X$?

These ideas can be written as the following Complete Search recurrences, `ways(N, K)`:

1. `ways(N, 1) = 1` // we can only use 1 number to add up to N

2. `ways(N, K) =` $\sum_{X=0}^{N}$ `ways(N - X, K - 1)` // sum all possible ways, recursively

This problem has overlapping sub-problems[23], but there are only $N \times K$ possible states of (N, K). The cost to compute one 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 is in `ways(N, K)`.

Note: This problem actually just need the result modulo 1 Million (i.e. the last 6 digits of the answer). See Section 5.5.7 that discuss modulo arithmetic computation.

Example codes: `ch3_11_UVa10943.cpp`; `ch3_11_UVa10943.java`

[23]Try this smallest test case $N = 1, K = 3$ with overlapping sub-problems: State $(N = 0, K = 1)$ is reached twice.

Remarks About Dynamic Programming in Programming Contests

There are other classical DP problems that we choose not to cover in this book as they are very rare such as Matrix Chain Multiplication [3], Optimal Binary Search Tree [24], etc. However, we will discuss Floyd Warshall's DP algorithm in Section 4.5; and discuss String Alignment (Edit Distance), Longest Common Subsequence (LCS), plus other DP on String in Section 6.5.

In the past (1990s), if a contestant is good with DP, he can become a 'king of programming contests' as usually DP problems are the 'decider problems'. But 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 by memorizing the solutions of classical DP problems! Try to go to the basics of problem solving using DP: Determine states (the table) that can uniquely and efficiently represent sub-problems and then how to fill that table, either via top-down recursion or bottom-up loop(s).

There is no better way to master all these problem solving paradigms other than solving real programming problems! Here, we give several examples. Then once you are familiar with the examples shown in this section, learn the newer forms of DP problems that keep appearing in recent programming contests. Some of them are discussed in Section 8.4.

Programming Exercises for solvable using Dynamic Programming:

- Longest Increasing Subsequence (LIS)
  1. UVa 00111 - History Grading (straight-forward, 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? *** (must use $O(n \log k)$ LIS, print solution)
  5. UVa 00497 - Strategic Defense Initiative (solution must be printed)
  6. UVa 10131 - Is Bigger Smarter? (sort elephants based on dec IQ; LIS on inc weight)
  7. UVa 10534 - Wavio Sequence (must use $O(n \log k)$ LIS twice)
  8. **UVa 11456 - Trainsorting *** (get max(LIS(i) + LDS(i) - 1), $\forall i \in [0 \ldots N\text{-}1]$)
  9. **UVa 11790 - Murcia's Skyline *** (combination of classical LIS+LDS, weighted)
  10. LA 2815 - Tiling Up Blocks (Kaohsiung03)
- Max Sum
  1. **UVa 00108 - Maximum Sum *** (max 2D sum, elaborated in this section)
  2. **UVa 00507 - Jill Rides Again *** (max 1D sum/max consecutive subsequence)
  3. UVa 00787 - Maximum Sub-sequence ... (max 1D *product*, careful with 0, BigInteger)
  4. UVa 00836 - Largest Submatrix (convert '0' to -INF, then run max 2D sum)
  5. UVa 00983 - Localized Summing for ... (2D sum, get submatrix)
  6. UVa 10074 - Take the Land (max 2D sum)
  7. UVa 10667 - Largest Block (max 2D sum)
  8. UVa 10684 - The Jackpot (max 1D sum/max consecutive subsequence)
  9. **UVa 10827 - Maximum Sum on a Torus *** (hint: copy N*N to N*2N matrix)
- 0-1 Knapsack (Subset Sum)
  1. UVa 00562 - Dividing Coins (use one dimensional table)
  2. UVa 00990 - Diving For Gold (print the solution too)
  3. UVa 10130 - SuperSale (discussed in this section)
  4. UVa 10261 - Ferry Loading (s: current car, left, right)
  5. **UVa 10616 - Divisible Group Sum *** (Subset Sum, input can be -ve, long long)
  6. UVa 10664 - Luggage (Subset Sum)
  7. **UVa 10819 - Trouble of 13-Dots *** (0-1 knapsack with 'credit card' twist!)
  8. UVa 11658 - Best Coalition (s: id, share; t: form/ignore coalition with person id)
  9. **UVa 11832 - Account Book *** (interesting DP[24]; print solution)
  10. LA 3619 - Sum of Different Primes (Yokohama06)

[24] s: id, val (use offset to handle negative numbers, see Section 8.4.6); t: plus or minus.

- 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 and UVa 674)
  4. UVa 00674 - Coin Change (discussed in this section)
  5. **UVa 10306 - e-Coins *** (coin change variant: each coin has two components)
  6. UVa 11137 - Ingenuous Cubrency (use long long)
  7. **UVa 11517 - Exact Change *** (a slight 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[25])
  3. **UVa 11284 - Shopping Trip *** (TSP variant; can go home early[26])
- Other Classical DP Problems in this Book
  1. **UVa 00348 - Optimal Array Mult ... *** (Matrix Chain Multiplication)
  2. **UVa 10304 - Optimal Binary Search Tree *** (as the name implies)

  Also see: Floyd Warshall's for All-Pairs Shortest Paths problem (see Section 4.5)

  Also see: String Alignment (Edit Distance) (see Section 6.5)

  Also see: Longest Common Subsequence (see Section 6.5)
- Non Classical (The Easier Ones)
  1. UVa 00116 - Unidirectional TSP (similar to UVa 10337)
  2. UVa 00571 - Jugs (the solution can be suboptimal)
  3. UVa 10003 - Cutting Sticks (discussed in this section)
  4. UVa 10036 - Divisibility (must use offset technique as val can be negative)
  5. **UVa 10337 - Flight Planner *** (DP; also solvable with Dijkstra's (Section 4.4.3))
  6. UVa 10400 - Game Show Math (backtracking with clever pruning is still sufficient)
  7. UVa 10465 - Homer Simpson (one dimensional DP table)
  8. **UVa 10721 - Bar Codes *** (s: n, k; t: try all from 1 to m)
  9. UVa 10910 - Mark's Distribution (two dimensional DP table)
  10. UVa 10912 - Simple Minded Hashing (s: len, last, sum; t: try next char)
  11. **UVa 10943 - How do you add? *** (s: n, k; t: try all the possible splitting points)
  12. UVa 11341 - Term Strategy (s: id, h_learned, h_left; t: learn module 'id' 1 hour/skip)
  13. UVa 11407 - Squares (can be memoized)
  14. UVa 11420 - Chest of Drawers (s: prev, id, numlck; lock/unlock this chest)
  15. UVa 11450 - Wedding Shopping (discussed in this section)
  16. UVa 11703 - sqrt log sin (can be memoized)

[25] Actually, since $N \leq 11$, this problem is still solvable with recursive backtracking and pruning.

[26] We just need to tweak the DP TSP recurrence a bit. At each state, we have one more option: go home early.

3.6 Chapter Notes

Many problems in ICPC or IOI require one or combination (see Section 8.2) of these problem solving paradigms. If we have to nominate a chapter in this book that contestants have to really master, we will choose this one.

The main source of the 'Complete Search' material in this chapter is the USACO training gateway [29]. We adopt the name 'Complete Search' rather than 'Brute-Force' as we believe that some Complete Search solution can be clever and fast enough, although it is complete. We believe the term 'clever Brute-Force' is a bit self-contradicting. We will discuss some more advanced search techniques later in Section 8.3, e.g. A* Search, Depth Limited Search (DLS), Iterative Deepening Search (IDS), Iterative Deepening A* (IDA*).

Divide and Conquer paradigm is usually used in the form of its popular algorithms: binary search and its variants, merge/quick/heap sort, and data structures: binary search tree, heap, segment tree, etc. We will see more D&C later in Computational Geometry (Section 7.4).

Basic Greedy and Dynamic Programming (DP) techniques techniques are always included in popular algorithm textbooks, e.g. Introduction to Algorithms [3], Algorithm Design [23], Algorithm [4]. However, to keep pace with the growing difficulties and creativity of these techniques, especially the DP techniques, we include more references from Internet: TopCoder algorithm tutorial [17] and recent programming contests. In this book, we will revisit DP again on four occasions: Floyd Warshall's DP algorithm (Section 4.5), DP on (implicit) DAG (Section 4.7.1), DP on String (Section 6.5), and More Advanced DP (Section 8.4).

However, for some real-life problems, especially those that are classified as NP-Complete [3], many of the approaches discussed so far will not work. For example, 0-1 Knapsack Problem which has $O(NS)$ DP complexity is too slow if S is big; TSP which has $O(N^2 \times 2^N)$ DP complexity is too slow if N is much larger than 16. For such problems, people use heuristics or local search: Tabu Search [15, 14], Genetic Algorithm, Ants Colony Optimization, Beam Search, etc.

There are $\approx$ **179 UVa (+ 15 others) programming exercises** discussed in this chapter.
(Only 109 in the first edition, a 78% increase).

There are **32 pages** in this chapter.
(Also 32 in the first edition, but some content have been reorganized to Chapter 4 and 8).

Chapter 4

Graph

We Are All Connected
— **Heroes TV Series**

4.1 Overview and Motivation

Many real-life problems can be classified as graph problems. Some have efficient solutions. Some do not yet have them. 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 tree, shortest paths, maximum flow, and discuss graphs with special properties.

In writing this chapter, we assume that the readers are *already* familiar with the following graph terminologies: Vertices/Nodes, Edges, Un/Weighted, Un/Directed, In/Out Degree, Self-Loop/Multiple Edges (Multigraph) versus Simple Graph, Sparse/Dense, Path, Cycle, Isolated versus Reachable Vertices, (Strongly) Connected Component, Sub-Graph, Complete Graph, Tree/Forest, Euler/Hamiltonian Path/Cycle, Directed Acyclic Graph, and Bipartite Graph. If you encounter any unfamiliar term, please read other reference books like [3, 32] (or browse Wikipedia) and search for that particular term.

We also assume that the readers have read various ways to represent graph information that have been discussed earlier in Section 2.3.1. That is, we will directly use the terms like: Adjacency Matrix, Adjacency List, Edge List, and implicit graph without redefining them. Please revise Section 2.3.1 if you are not familiar with these graph data structures.

Our research so far on graph problems in recent ACM ICPC regional contests (especially in Asia) 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 has only 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 all these materials.

For IOI, the syllabus [10] 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 assists 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. Our DFS implementation uses the help of a *global* vector of integer: `vi dfs_num` to distinguish the state of each vertex between 'unvisited' (we use a constant value `DFS_WHITE = -1`) and 'visited' (we use another constant value `DFS_BLACK = 1`). Initially, all values in `dfs_num` are set to 'unvisited'.

Calling `dfs(u)` starts DFS from a vertex u, mark vertex u as 'visited', and then DFS recursively visits each 'unvisited' neighbor v of u (i.e. edge $u - v$ exists in the graph). DFS will explore the graph 'depth-first' and will 'backtrack' and explore another branch only after it has no more options in the current branch to visit. The snippet of DFS code is shown below:

```
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 shortcuts are useful in competitive programming

vi dfs_num;                 // for simplicity, this variable is set to be global

void dfs(int u) {          // DFS for normal usage: as graph traversal algorithm
  dfs_num[u] = DFS_BLACK;      // important step: 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] == DFS_WHITE)         // important check to avoid cycle
      dfs(v.first);      // recursively visits unvisited neighbors v of vertex u
} }
```

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.

On the sample graph in Figure 4.1, `dfs(0)` – calling DFS from a start 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. Note that this sequence of visitation depends very much on the way we order the neighbors of a vertex, i.e. the sequence $0 \rightarrow 1 \rightarrow 3 \rightarrow 2$ (backtrack to 3) $\rightarrow 4$ is also a possible visitation sequence. For simplicity, we usually just order the vertices based on their vertex numbers.

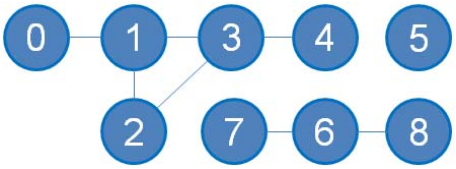


Figure 4.1: Sample Graph

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)`.

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. Backtracking un-flag visited vertices when it backtracks. By not revisiting vertices, DFS runs in $O(V + E)$, but the time complexity of backtracking is exponential.

```
void backtracking(state) {
  if (hit end state or invalid state)   // we need terminating/pruning condition
    return;                           // to avoid cycling and to speed up search
  for each neighbor of this state                        // try all permutation
    backtracking(neighbor);
}
```

4.2.2 Breadth First Search (BFS)

Breadth First Search - abbreviated as BFS - is another graph traversal algorithm. BFS starts with the insertion of 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, 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. Thus the name is *breadth-first*. BFS algorithm also runs in $O(V+E)$ and $O(V^2)$ on a graph represented using an Adjacency List and Adjacency Matrix, respectively.

Implementing BFS is easy if we utilize C++ STL or Java API. We use `queue` to order the sequence of visitation and `map` 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 feature is used later to solve a special case of Single-Source Shortest Paths problem (see Section 4.4).

```
// inside int main() -- no recursion, so we do not need to use separate function
  map<int, int> dist; dist[s] = 0; // distance from source s to s is 0 (default)
  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 neighbors of u
      if (!dist.count(v.first)) {    // dist.find(v.first) == dist.end() also OK
        dist[v.first] = dist[u] + 1;                        // v unvisited + reachable
        q.push(v.first);                                    // enqueue v for next step
  } } }
```

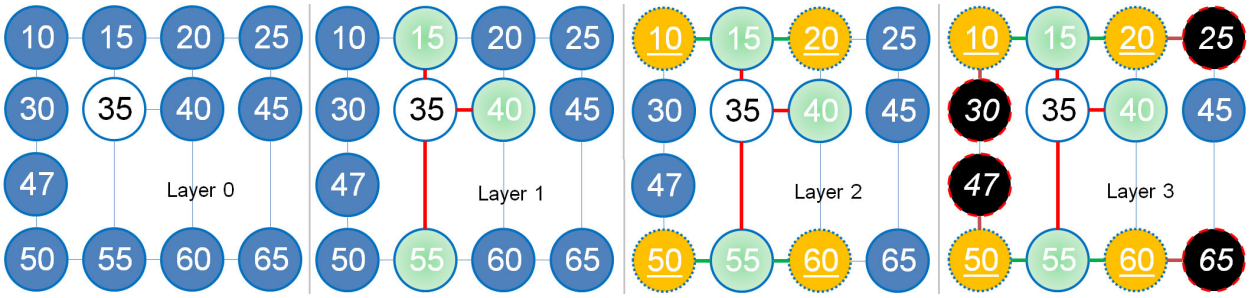


Figure 4.2: Animation of BFS (from UVa 336 [28])

If we run BFS from the vertex labeled with 35 (i.e. the source vertex $s = 35$) on the connected undirected graph shown in Figure 4.2, we will visit the vertices in the following order:

```
Layer 0:, visit 35
Layer 1:, visit 15, visit 55, visit 40
Layer 2:, visit 10, visit 20, visit 50, visit 60
Layer 3:, visit 30, visit 25, visit 47, visit 65
Layer 4:, visit 45 (not yet shown in Figure 4.2 above)
```

Exercise 4.2.2.1: This implementation uses `map<VERTEX-TYPE, int> dist` to store distance information. This may be useful if `VERTEX-TYPE` is not an integer, e.g. a pair `ii` of (position, bitmask) in UVa 321 - The New Villa, `pair<ii, ii>` of ((row, col), (direction, color)) in UVa 10047 - The Monocycle, etc. However, this trick adds a $\log V$ factor to the $O(V + E)$ BFS complexity, making it $O((V + E) \log V)$. Please rewrite this implementation to use `vector<int> dist` instead!

Exercise 4.2.2.2: Why does 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?

4.2.3 Finding Connected Components (in an Undirected Graph)

DFS and BFS are not just useful for traversing a graph. They can be used to solve many other graph problems. The first few applications below can be solved with *either* DFS or BFS although some of the last few applications 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 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.

```
// inside int main() -- this is the DFS solution
  numCC = 0;
  dfs_num.assign(V, DFS_WHITE);    // this sets all vertices' state to DFS_WHITE
  for (int i = 0; i < V; i++)                  // for each vertex i in [0..V-1]
    if (dfs_num[i] == DFS_WHITE)            // if that vertex is not visited yet
      printf("Component %d:", ++numCC), dfs(i), printf("\n");   // 3 lines here!
  printf("There are %d connected components\n", numCC);

// For the sample graph in Figure 4.1, the output is like this:
// Component 1: 0 1 2 3 4
// Component 2: 5
// Component 3: 6 7 8
// There are 3 connected components
```

Exercise 4.2.3.1: We can also use Union-Find Disjoint Sets data structure (see Section 2.3.2) 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 `dfs(u)` (or `bfs(u)`) can be used to *label* (or in this case 'color') and count the size of each component. This variant is more famously known as 'flood fill' and usually performed on *implicit* graph (usually a 2D grid).

```
int dr[] = {1,1,0,-1,-1,-1, 0, 1};              // S,SE,E,NE,N,NW,W,SW neighbors
int dc[] = {0,1,1, 1, 0,-1,-1,-1}; // trick to explore an implicit 2D grid graph

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 the grid, prune
  if (grid[r][c] != c1) return 0;   // this vertex does not have color c1, prune
  int ans = 1;         // add 1 to ans because vertex (r, c) has c1 as its color
  grid[r][c] = c2;          // now recolor vertex (r, c) to c2 to avoid cycling!
  for (int d = 0; d < 8; d++)  // recurse to neighbors, see how neat the code is
    ans += floodfill(r + dr[d], c + dc[d], c1, c2);  // with help of dr[] + dc[]
  return ans;
}
```

Let's see an example below. The implicit graph is a 2D grid where the vertices are the cells in the grid and the edges are the connection 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 count the size of) a wet area by using floodfill. Here is an example of running floodfill from row 2, column 1 (0-based indexing), replacing 'W' to '.':

```
// inside int main()
// read the grid as a global 2D array and also read (row, col) query coordinate
  printf("%d\n", floodfill(row, col, 'W', '.')); // label+count size of wet area

// LLLLLLLLL         LLLLLLLLL
// LLWWLLWLL         LL..LLWLL // The size of connected component (the connected
// LWWLLLLLL (R2,C1) L..LLLLLL //  'W's) with one 'W' at (row 2, column 1) is 12
// LWWWLWWLL         L...L..LL //       Notice that all these connected 'W's are
// LLLWWWLLL ======> LLL...LLL //             replaced with '.'s after floodfill
// LLLLLLLLL         LLLLLLLLL
// LLLWWLLWL         LLLWWLLWL
// LLWLWLLLL         LLWLWLLLL
// LLLLLLLLL         LLLLLLLLL
```

4.2.5 Topological Sort (of a 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 $(u \rightarrow v)$ exists in the DAG. Every DAG has one *or more* topological sorts.

One application of topological sorting is to find a possible sequence of modules that a University student has to take to fulfill his graduation requirement. Each module has certain pre-requisites to be met. This pre-requisites 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 constraint.

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 topoSort;              // global vector to store the toposort in reverse order

void dfs2(int u) {    // change function name to differentiate with original dfs
  dfs_num[u] = DFS_BLACK;
  for (int j = 0; j < (int)AdjList[u].size(); j++) {
    ii v = AdjList[u][j];
    if (dfs_num[v.first] == DFS_WHITE)
      dfs2(v.first);
  }
  topoSort.push_back(u); }                  // that's is, this is the only change

// inside int main()
  topoSort.clear();
  memset(dfs_num, DFS_WHITE, sizeof dfs_num);
  for (int i = 0; i < V; i++)             // this part is the same as finding CCs
    if (dfs_num[i] == DFS_WHITE)
      dfs2(i);
  reverse(topoSort.begin(), topoSort.end());                  // reverse topoSort
  for (int i = 0; i < (int)topoSort.size(); i++)       // or you can simply read
    printf(" %d", topoSort[i]);           // the content of 'topoSort' backwards
  printf("\n");
// For the sample graph in Figure 4.3, 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 list of vertices explored only after visiting all the subtrees below u. As `vector` only support *efficient insertion* from back, we 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 more constant operation.

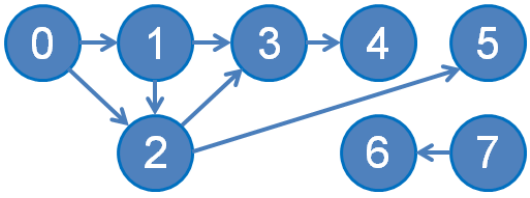


Figure 4.3: Example of Toposort on DAG

Exercise 4.2.5.1: Do you understand why appending `topoSort.push_back(u)` in the standard DFS code is enough to help us find topological sort of a DAG? Explain!

Exercise 4.2.5.2: Implement the toposort variant below (a 'modified BFS') to solve UVa 11060.

```
enqueue vertices with zero incoming degree into a (priority) queue;
while (queue is not empty) {
  vertex u = queue.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, queue.enqueue(v);
}
```

4.2.6 Bipartite Graph Check

Bipartite graph has important applications that we will see later in Section 4.7.4. Here, we just want to know if a graph is bipartite (a.k.a 2/bi-colorable). We can use BFS or DFS to check whether a graph is bipartite, but we feel that BFS is more natural for such check, as shown below:

```
// inside int main()
  queue<int> q; q.push(s);
  map<int, int> dist; dist[s] = 0;
  bool isBipartite = true;       // addition of one boolean flag, initially true

  while (!q.empty()) {                    // 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 (!dist.count(v.first)) {
        dist[v.first] = 1 - dist[u];   // but now, instead of recording distance
        q.push(v.first); }                   // we just record two colors {0, 1}
      else if (dist[v.first] == dist[u])     // if u-v is neighbor and both have
        isBipartite = false;        // the same color, we have coloring conflict
  } }
```

Exercise 4.2.6.1: Implement bipartite check using DFS instead!

Exercise 4.2.6.2: Is this statement true: "Bipartite graph has no odd cycle"?

4.2.7 Graph Edges Property Check via DFS Spanning Tree

Running DFS on a connected component of a graph forms a DFS *spanning tree*[1] (or *spanning forest* if the graph has more than one component and DFS is run per component). With one more vertex state: `DFS_GRAY = 2` (visited *but not yet completed*) on top of `DFS_BLACK` (visited *and completed*), we can use this DFS spanning tree (or forest) to classify graph edges into three types:

1. Tree edges: those traversed by DFS, i.e. from vertex with `DFS_GRAY` to vertex with `DFS_WHITE`.
2. Back edges: part of a cycle, i.e. from vertex with `DFS_GRAY` to vertex with `DFS_GRAY` too.
   This is probably the most frequently used application of this algorithm.
   Note that usually we 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 `DFS_GRAY` to vertex with `DFS_BLACK`.
   These two type of edges are not typically used in programming contest problems.

Figure 4.4 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 $1 \rightarrow 2 \rightarrow 3 \rightarrow 1$ is a (true) cycle and we classify edge $(3 \rightarrow 1)$ as a back edge, whereas $0 \rightarrow 1 \rightarrow 0$ is not a cycle but it is just a bi-directional edge (0-1). The code for this DFS variant is shown below.

Exercise 4.2.7.1: Perform graph edges property check on the graph in Figure 4.8. Assume that you start DFS from vertex 0. How many back edges that you can find this time?

```
// inside int main()
  dfs_num.assign(V, DFS_WHITE); dfs_parent.assign(V, 0);
  for (int i = 0; i < V; i++)
    if (dfs_num[i] == DFS_WHITE)
      printf("Component %d:\n", ++numComp), graphCheck(i);     // 2 lines in one
```

[1] A spanning tree T of a connected undirected graph G is a tree with all vertices but subset of the edges of G.

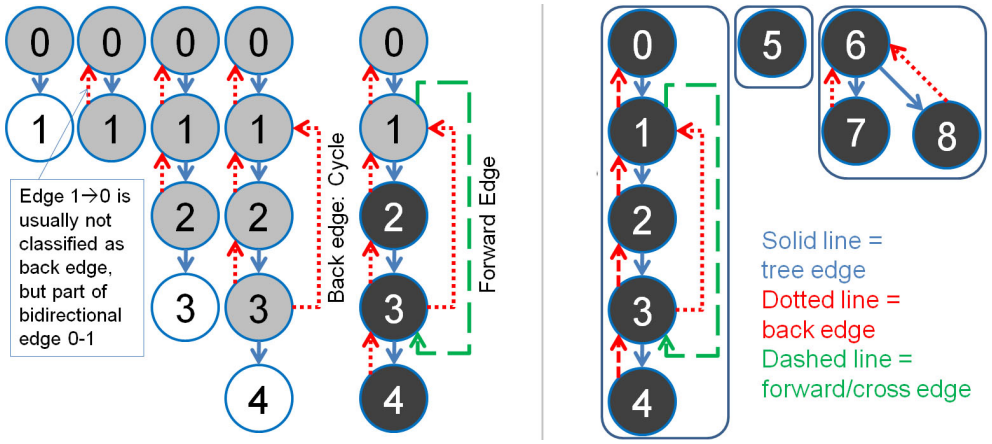


Figure 4.4: Animation of DFS when Run on the Sample Graph in Figure 4.1

```
void graphCheck(int u) {                  // DFS for checking graph edge properties
  dfs_num[u] = DFS_GRAY;   // color this as DFS_GRAY (temp) instead of DFS_BLACK
  for (int j = 0; j < (int)AdjList[u].size(); j++) {
    ii v = AdjList[u][j];
    if (dfs_num[v->first] == DFS_WHITE) {    // Tree Edge, DFS_GRAY to DFS_WHITE
      dfs_parent[v->first] = u;                 // parent of this children is me
      graphCheck(v->first);
    }
    else if (dfs_num[v->first] == DFS_GRAY) {            // DFS_GRAY to DFS_GRAY
      if (v->first == dfs_parent[u])         // to differentiate these two cases
        printf(" Bidirectional (%d, %d)-(%d, %d)\n", u, v->first, v->first, u);
      else  // the most frequent application: check if the given graph is cyclic
        printf(" Back Edge (%d, %d) (Cycle)\n", u, v->first);
    }
    else if (dfs_num[v->first] == DFS_BLACK)            // DFS_GRAY to DFS_BLACK
      printf(" Forward/Cross Edge (%d, %d)\n", u, v->first);
  }
  dfs_num[u] = DFS_BLACK;     // after recursion, color this as DFS_BLACK (DONE)
}

// For the sample graph in Figure 4.1, the output is like this:
// Component 1:
//  Bidirectional (1, 0) - (0, 1)
//  Bidirectional (2, 1) - (1, 2)
//  Back Edge (3, 1) (Cycle)
//  Bidirectional (3, 2) - (2, 3)
//  Bidirectional (4, 3) - (3, 4)
//  Forward/Cross Edge (1, 4)
// Component 2:
// Component 3:
//  Bidirectional (7, 6) - (6, 7)
//  Bidirectional (8, 6) - (6, 8)
```

4.2.8 Finding Articulation Points and Bridges (in an Undirected Graph)

Motivating problem: Given a road map (undirected graph) with costs associated to all intersections (vertices) and roads (edges), sabotage either a single intersection or a single road that has minimum cost such that the road network breaks down. 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[2] 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 (although they are still well defined for directed graphs).

A naïve algorithm to find articulation points is shown below (can be tweaked to find bridges too):

1. Run $O(V+E)$ DFS (or BFS) to count number of connected components of the original graph
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) to check if number of connected components 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 problem 22.2 in [3]), is just another extension from the previous DFS code shown earlier.

This algorithm maintains 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 `DFS_WHITE` versus `DFS_GRAY`/`DFS_BLACK`). The other number `dfs_low(u)` stores the lowest `dfs_num` reachable from DFS spanning sub tree of u. Initially `dfs_low(u) = dfs_num(u)` when vertex u is first visited. Then, `dfs_low(u)` can only be made smaller if there is a cycle (*one* 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.

See Figure 4.5 for clarity. In these two sample graphs, we run `articulationPointAndBridge(0)`. Suppose for the graph in Figure 4.5 – 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`.

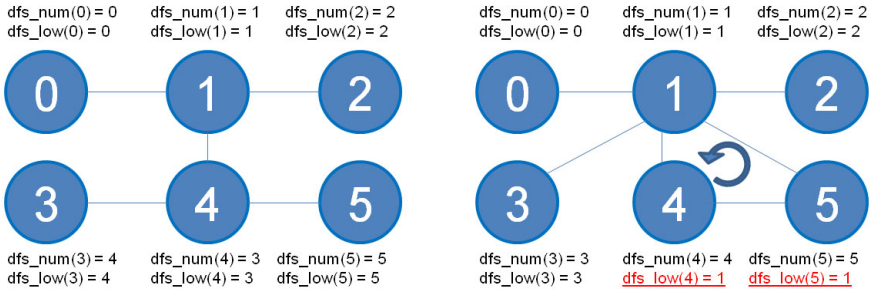


Figure 4.5: Introducing two More DFS Attributes: `dfs_num` and `dfs_low`

Suppose for the graph in Figure 4.5 – 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). Here, 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)` $\geq$ `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 connected to vertex v that can reach vertex w with a lower `dfs_num(w)` (which further implies that w is the ancestor of u in the DFS spanning tree). Thus, to reach that ancestor of u from v, one *must* pass through vertex u. This implies that removing the 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 (a trivial case that is not detected by this algorithm).

[2]All edges incident to this vertex are also removed.

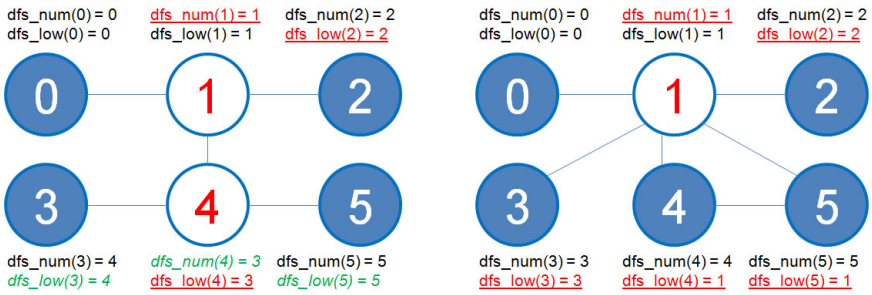


Figure 4.6: Finding Articulation Points with `dfs_num` and `dfs_low`

See Figure 4.6 for more details. On the graph in Figure 4.6 – left side, vertices 1 and 4 are articulation points, because for example in edge 1-2, we see that `dfs_low(2)` $\geq$ `dfs_num(1)` and in edge 4-5, we also see that `dfs_low(5)` $\geq$ `dfs_num(4)`. On the graph in Figure 4.6 – right side, only vertex 1 is the articulation point, because for example in edge 1-5, `dfs_low(5)` $\geq$ `dfs_num(1)`.

The process to find bridges is similar. When `dfs_low(v)` $>$ `dfs_num(u)`, then edge u-v is a bridge. In Figure 4.7, 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 – edge 4-5, `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).

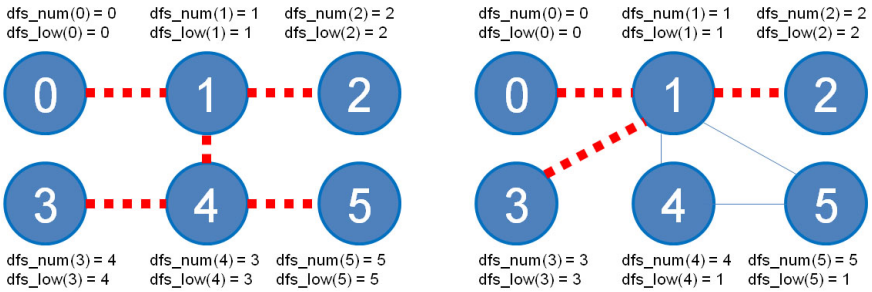


Figure 4.7: Finding Bridges, also with `dfs_num` and `dfs_low`

The snippet of the code for this algorithm is as follow:

```
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] == DFS_WHITE) {                          // a tree edge
      dfs_parent[v.first] = u;
      if (u == dfsRoot) rootChildren++;  // special case, count children of 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, DFS_WHITE); 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] == DFS_WHITE) {
      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);

// For the sample graph in Figure 4.5/4.6/4.7 RIGHT, the output is like this:
// Bridges:
//  Edge (1, 2) is a bridge
//  Edge (1, 3) is a bridge
//  Edge (0, 1) is a bridge
// Articulation Points:
//  Vertex 1
```

Exercise 4.2.8.1: Examine the graph in Figure 4.1 without running the algorithm discussed here. Which vertices are articulation points and which edges are bridges?

4.2.9 Finding Strongly Connected Components (in a Directed Graph)

Yet another application of DFS is to find *strongly* connected components in a *directed* graph. This is a different problem to finding connected components in an undirected graph. In Figure 4.8, we have a similar graph to the graph in Figure 4.1, but now the edges are directed. Although the graph in Figure 4.8 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 4.8, as highlighted with red boxes: {0}, {1, 3, 2}, and {4, 5, 7, 6}. Note: If these SCCs are contracted (shrunk into larger vertices), they form a DAG. This will be discussed again in Section 8.2.3.

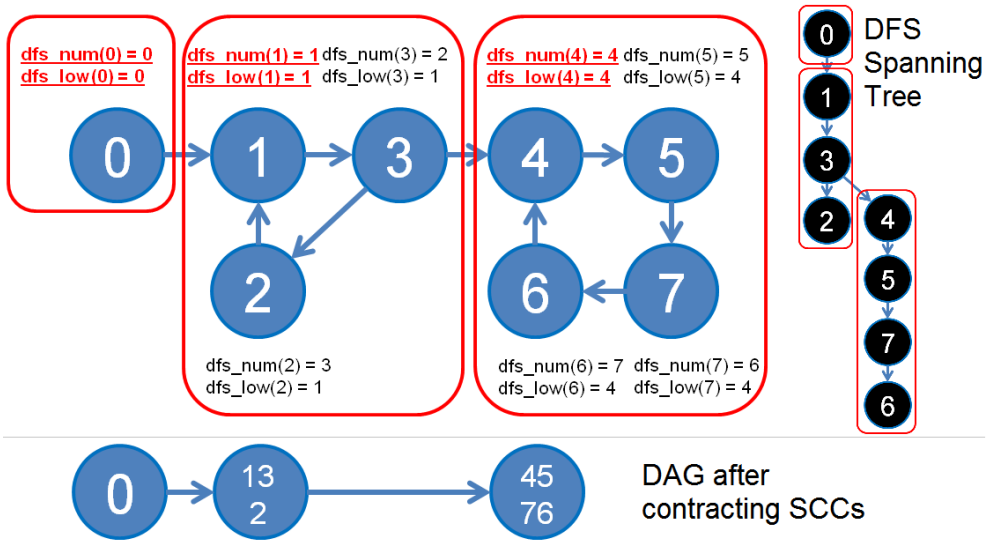


Figure 4.8: An Example of a Directed Graph and its Strongly Connected Components (SCC)

There are at least two known algorithms to find SCCs: Kosaraju's – explained in [3] and Tarjan's algorithm. In this book, we adopt Tarjan's version, as it extends naturally from our previous discussion of finding Articulation Points and Bridges – also due to Tarjan.

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 4.8). Vertex u in this DFS spanning tree with `dfs_low(u) = dfs_num(u)` is the root (start) of an SCC (observe vertex 0 (SCC with 1 member only), 1 (plus 3, 2), and 4 (plus 5, 7, 6) in Figure 4.8). The code below explores the directed graph and reports 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] == DFS_WHITE)
      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, DFS_WHITE); dfs_low.assign(V, 0); visited.assign(V, 0);
  dfsNumberCounter = numSCC = 0;
  for (int i = 0; i < V; i++)
    if (dfs_num[i] == DFS_WHITE)
      tarjanSCC(i);

// For the sample graph in Figure 4.8, the output is like this:
// SCC 1: 6 7 5 4
// SCC 2: 2 3 1
// SCC 3: 0
```

Exercise 4.2.9.1: Study **Kosaraju's** algorithm for finding Strongly Connected Component (discussed in [3]) and then decide which one is simpler: Tarjan's or Kosaraju's?

Exercise 4.2.9.2: Is this statement true: "If two vertices are in the same SCC, then there is no path between them that ever leaves the SCC."?

Exercise 4.2.9.3: Write a code that takes in a Directed Graph and then convert it into a Directed Acyclic Graph (DAG) by contracting the SCCs.

Example codes: `ch4_01_dfs.cpp; ch4_02_UVa469.cpp`; `ch4_01_dfs.java; ch4_02_UVa469.java`

Remarks About DFS and BFS for 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 algorithms on top of their basic form for traversing a graph. In ICPC, any of these variants 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 flood fill problems is getting smaller.

Topological sort is rarely used per se, but it is 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 version presented as exercise in this section (the 'modified BFS' that only enqueue vertices with 0-incoming degrees) is also 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 8.2.3. 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 in IOI 2009 syllabus [10].

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 solve using the recursive and more memory friendly DFS. We do not normally use BFS for pure graph traversal but we will use it to solve the Single-Source Shortest Paths problem on unweighted graph (see Section 4.4). Table 4.1 shows important comparison between these two popular graph traversal algorithms.

	$O(V+E)$ **DFS**	$O(V+E)$ **BFS**
Pros	Uses less memory (good for large graph)	Can solve SSSP on unweighted graphs
Cons	Cannot solve SSSP on unweighted graphs	Uses more memory (bad for large graph)
Code	Slightly easier to code	Just a bit longer to code

Table 4.1: Graph Traversal Algorithm Decision Table

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 Minotaur *** (AdjMat, parsing, traversal)
  3. UVa 00280 - Vertex (graph, reachability test by traversing the graph)
  4. UVa 00614 - Mapping the Route (traversal on *implicit* graph)
  5. UVa 00824 - Coast Tracker (traversal on *implicit* graph)
  6. UVa 10113 - Exchange Rates (just graph traversal, but uses fraction and gcd)
  7. UVa 10116 - Robot Motion (traversal on *implicit* graph)
  8. UVa 10377 - Maze Traversal (traversal on *implicit* graph)
  9. UVa 10687 - Monitoring the Amazon (build dir graph with geo, reachability test)
  10. **UVa 11831 - Sticker Collector Robot *** (*implicit* graph; input order is 'NSEW'!)
  11. **UVa 11902 - Dominator *** (disable vertex one by one, check[3])
  12. UVa 11906 - Knight in a War Grid (BFS for reachability, several tricky cases[4])
  13. LA 3138 - Color a Tree (Beijing04, DFS)
  14. IOI 2011 - Tropical Garden (graph traversal; DFS; involving cycle)

[3]check if the reachability from vertex 0 changes.

[4]Be careful when M = 0 || N = 0 || M = N.

- Flood Fill/Finding Connected Components
  1. UVa 00260 - Il Gioco dell'X (6 neighbors per cell!)
  2. UVa 00352 - Seasonal War (count number of connected components (CC))
  3. UVa 00469 - Wetlands of Florida (count size of a CC; discussed in this section)
  4. UVa 00572 - Oil Deposits (count number of CCs, similar to UVa 352)
  5. UVa 00657 - The Die is Cast (there are three 'colors' here)
  6. UVa 00776 - Monkeys in a Regular Forest (label CCs with indices, format output)
  7. UVa 00782 - Countour Painting (replace ' ' with '#' whenever we are inside the grid)
  8. UVa 00784 - Maze Exploration (very similar with UVa 782)
  9. UVa 00785 - Grid Colouring (also very similar with UVa 782)
  10. UVa 00852 - Deciding victory in Go (interesting application to board game 'Go')
  11. UVa 00871 - Counting Cells in a Blob (find the size of the largest CC)
  12. UVa 10336 - Rank the Languages (count and rank CCs with similar color)
  13. UVa 10946 - You want what filled? (find CCs and rank them by their size)
  14. **UVa 11094 - Continents *** (tricky flood fill as it involves scrolling)
  15. **UVa 11110 - Equidivisions *** (flood fill + satisfy the constraints given)
  16. UVa 11244 - Counting Stars (count number of CCs)
  17. UVa 11470 - Square Sums (you can do 'flood fill'[5] layer by layer)
  18. UVa 11518 - Dominos 2 (unlike UVa 11504, we can treat the SCCs as simple CCs)
  19. UVa 11561 - Getting Gold (flood fill with extra blocking constraint)
  20. UVa 11749 - Poor Trade Advisor
  21. **UVa 11953 - Battleships *** (interesting twist of flood fill problem)
  22. LA 2817 - The Suspects (Kaohsiung03, Connected Components)
- 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 discussed in this section)
  5. **UVa 11060 - Beverages *** (must use the 'modified BFS' topological sort)
  6. UVa 11686 - Pick up sticks (toposort + cycle check)
     Also see: DP on (implicit/explicit) 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 check, take max(left, right))
  3. **UVa 11080 - Place the Guards *** (bipartite graph check, some tricky cases)
  4. **UVa 11396 - Claw Decomposition *** (eventually 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)
- Finding Strongly Connected Components
  1. **UVa 00247 - Calling Circles *** (SCC + printing solution)
  2. UVa 10731 - Test (SCC + printing solution)
  3. **UVa 11504 - Dominos *** (count |SCCs| without in-degree from node outside SCC)
  4. UVa 11709 - Trust Groups (find number of SCC)
  5. UVa 11770 - Lighting Away (similar to UVa 11504)
  6. **UVa 11838 - Come and Go *** (check if the input graph is strongly connected)
  7. LA 4099 - Sub-dictionary (Iran07, SCC)

[5]There is other way to solve this problem, e.g. by finding the patterns.

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 4.9), 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!

To satisfy the connectivity criteria, edges in E' must form a *tree* that spans (covers) all $V \in G$ – the *spanning tree*! There can be several valid spanning trees in G, i.e. see Figure 4.9, middle and right sides. One of them is the required solution that satisfies the minimal weight criteria.

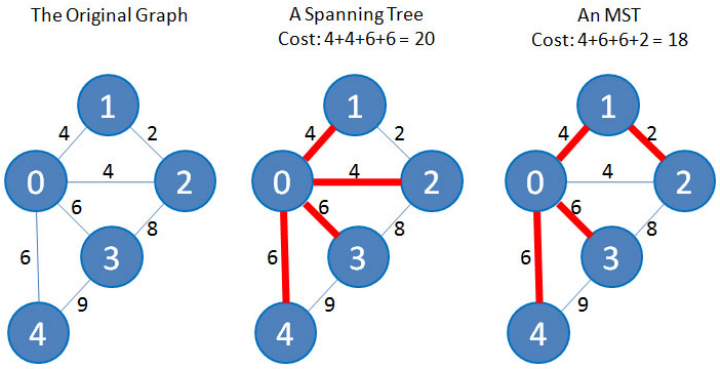


Figure 4.9: Example of an MST Problem

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 networks 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 networks that connects all these villages.

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 [3, 32, 24, 34, 26, 1, 23, 4].

4.3.2 Kruskal's Algorithm

Joseph Bernard *Kruskal* Jr.'s algorithm first sort E edges based on non decreasing weight. This can be easily done by storing the edges in an EdgeList data structure (see Section 2.3.1) and then sort the edges. Then, Kruskal's algorithm *greedily* try to add each edge to 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 2.3.2. The code is short and in overall runs in O(sorting + trying to add each edge $\times$ cost of Union and Find operations) $= O(E \log E + E \times 1) = O(E \log E) = O(E \log V^2) = O(2 \times E \log V) = O(E \log V)$.

```
// inside int main()
  vector< pair<int, ii> > EdgeList;  // format: weight, two vertices of the edge
  for (int i = 0; i < E; i++) {
    scanf("%d %d %d", &a, &b, &weight);       // read the triple: (a, b, weight)
    EdgeList.push_back(make_pair(weight, ii(a, b))); }  // store: (weight, a, b)
  sort(EdgeList.begin(), EdgeList.end());   // sort by edge weight in O(E log E)

  int mst_cost = 0; initSet(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 (!isSameSet(front.second.first, front.second.second)) {    // if no cycle
      mst_cost += front.first;                      // add the weight of e to MST
      unionSet(front.second.first, front.second.second);       // link endpoints
  } }                             // note: the runtime cost of UFDS is very light

  // note: the number of disjoint sets must eventually be one for a valid MST
  printf("MST cost = %d (Kruskal's)\n", mst_cost);
```

Figure 4.10 shows the execution of Kruskal's algorithm on the graph shown in Figure 4.9, leftmost.

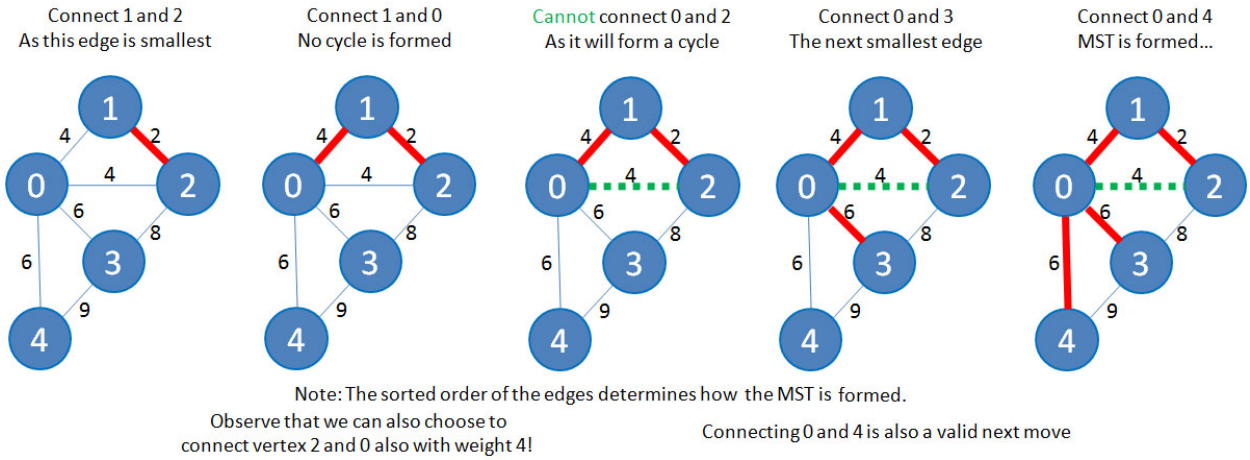


Figure 4.10: Animation of Kruskal's Algorithm for an MST Problem

Exercise 4.3.2.1: The given code above only stop after the last edge in EdgeList is processed. For some cases, we can stop Kruskal's algorithm earlier. Modify the given code to implement this!

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 enqueue a pair of information into a priority queue: The weight w and the other end point u of the edge $0-u$ 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* select 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 $u - v$ with weight w' that is incident to u is enqueued into the priority queue. 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(process each edge once $\times$ 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

void process(int vtx) {
  taken[vtx] = 1;
  for (int j = 0; j < 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 by using -ve sign to reverse order

// inside int main() --- assume the graph has been stored in AdjList
  taken.assign(V, 0);
  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 and process all edges incident to u
  }                                                  // each edge is in pq only once!
  printf("MST cost = %d (Prim's)\n", mst_cost);
```

Figure 4.11 shows the execution of Prim's algorithm on the same graph shown in Figure 4.9, leftmost. Please compare it with Figure 4.10 to study the similarities and differences between Kruskal's and Prim's algorithms.

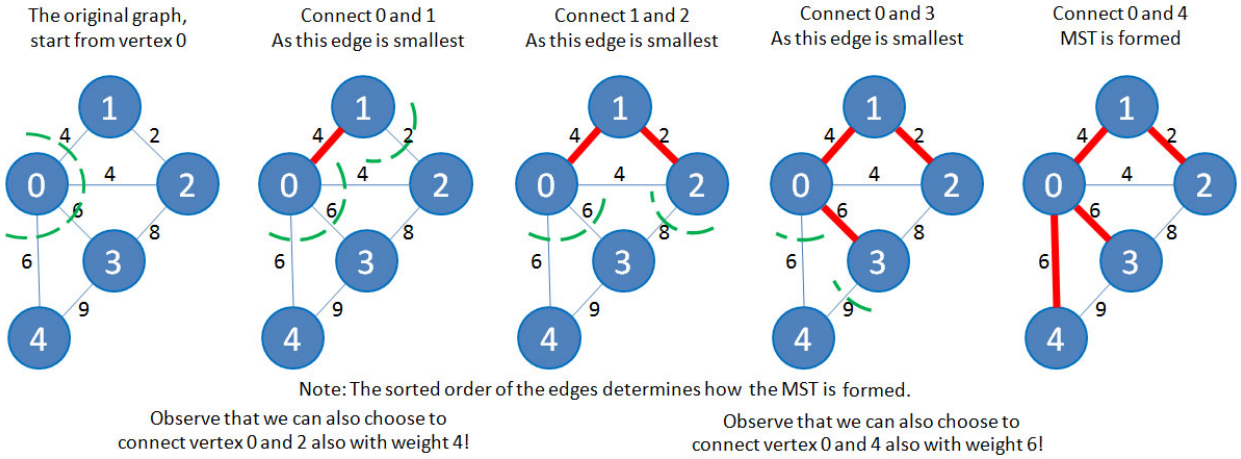


Figure 4.11: Animation of Prim's Algorithm for the Same MST Problem as in Figure 4.9, left

Example codes: `ch4_03_kruskal_prim.cpp`; `ch4_03_kruskal_prim.java`

4.3.4 Other Applications

Variants of basic MST problems are interesting. In this section, we will explore some of them.

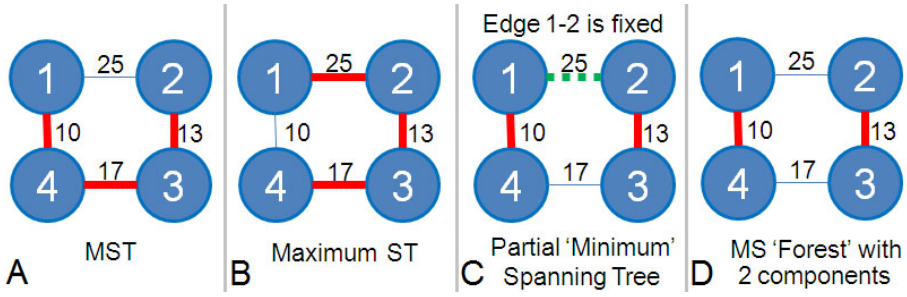


Figure 4.12: From left to right: MST, 'Maximum' ST, Partial 'Minimum' ST, MS 'Forest'

'Maximum' Spanning Tree

This is a simple variant where we want the maximum, instead of the minimum ST. In Figure 4.12.B, we see an example of a Maximum ST. Compare it with the corresponding MST (Figure 4.12.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.

Partial 'Minimum' Spanning Tree

In this variant, we do not start with a clean slate. Some edges in the given graph are already fixed and must be taken as part of the Spanning Tree solution. We must continue building the 'M'ST from there, thus the resulting Spanning Tree may turn out not to be an MST. That's why we put the term 'Minimum' in quotes. In Figure 4.12.C, we see an example when one edge 1-2 is already fixed. The actual MST is 10+13+17 = 40 which omits the edge 1-2 (Figure 4.12.A). However, the solution for this problem must be (25)+10+13 = 48 which uses the edge 1-2.

The solution for this variant is simple. After taking into account all the fixed edges, we continue running Kruskal's algorithm on the remaining free edges.

Minimum Spanning 'Forest'

In this variant, we want the spanning criteria, i.e. all vertices must be covered by some edges, but we can stop even though the spanning tree has not been formed as long as the spanning criteria is satisfied! This can happen when we have a spanning 'forest'. Usually, the desired number of components is told beforehand in the problem description. In Figure 4.12.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 components, then the solution is just 10+13 = 23 on Figure 4.12.D. That is, we omit the edge 3-4 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, we can terminate the algorithm.

Second Best Spanning Tree

Sometimes, we are interested to have alternative solutions. 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. Figure 4.13 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[6] edge is added into the MST. Here, edge 4-5 is taken out and edge 2-5 is added in.

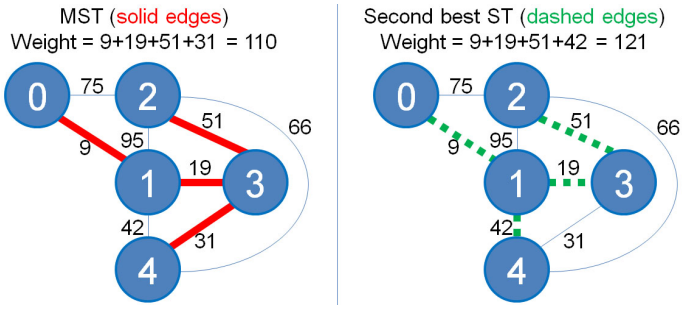


Figure 4.13: Second Best ST (from UVa 10600 [28])

A solution is to 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. The best spanning tree found after this process is the second best ST. Figure 4.14 shows this algorithm on the given graph. In overall, this algorithm runs in O(sort the edges + find original MST + find second best ST) = $O(E \log V + E + VE) = O(VE)$.

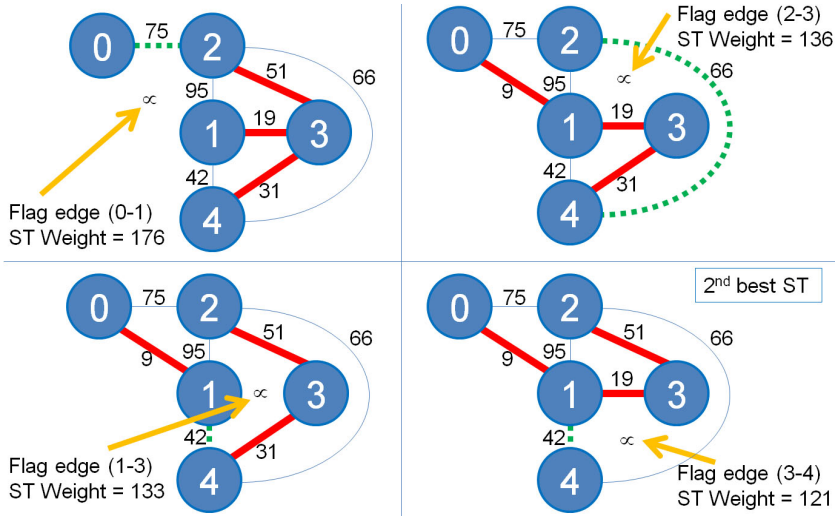


Figure 4.14: Finding the Second Best Spanning Tree from the MST

Exercise 4.3.4.1: Solve the four variants above ('Max' ST, Partial 'Min' ST, MS 'Forest', and Second Best ST) using Prim's algorithm instead. Which variant(s) is/are not Prim's-friendly?

[6] A chord edge is defined as an edge in graph G that is not selected in the MST of G.

Minimax (and Maximin)

The minimax path problem[7] is a problem of finding the minimum of maximum edge weight among all possible paths between two vertices i to j. 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 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 maximum edge weight along the only unique path between vertex i and j in this MST.

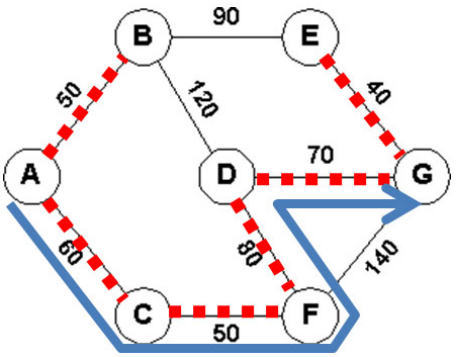


Figure 4.15: Minimax (UVa 10048 [28])

The overall time complexity is O(build MST + one traversal on the resulting tree). In Section 4.7.2, we will see that any traversal on tree is just $O(V)$. Thus the complexity of this approach is $O(E \log V + V) = O(E \log V)$.

In Figure 4.15, we have a graph with 7 vertices and 9 edges. The 6 chosen edges of the MST are shown as dotted lines in Figure 4.15. Now, if we are asked to find the minimax path between vertex 'A' and 'G' in Figure 4.15, we simply traverse the MST from vertex 'A' to 'G'. There will only be one way, path: A-C-F-D-G. The maximum edge weight found along the path is the required minimax cost: 80 (due to edge F-D).

Exercise 4.3.4.2: Solve the **maximin** path problem using the idea shown above!

Remarks About Minimum Spanning Tree in Programming Contests

To solve many MST problems in today's programming contests, we can rely on Kruskal's 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 2.3.2) to check for cycles. However, as we do love choices, we also discuss the other popular algorithm for MST: the 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, 10034, 11631), but the easy variant of 'Maximum' ST is also possible (UVa 10842, LA 4110). Notice that almost 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 partial 'Minimum' ST (UVa 10147, 10397), MS 'Forest' (UVa 10369, LA 3678), Second best ST (UVa 10600), Minimax/Maximin (UVa 544, 10048) are actually rare.

Nowadays the more general trend for MST problems is for the problem setters to write the MST problem in such a way that it is not clear that the problem is actually an MST problem (e.g. LA 3678, 4110, 4138). However, once the contestants spot this, the problem may become 'easy'.

Profile of Algorithm Inventors

Joseph Bernard Kruskal, Jr. (1928-2010) was an American computer scientist. His best known work 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.

[7]There can be many paths from 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.

Programming Exercises related to Minimum Spanning Tree:

- Standard
  1. UVa 00908 - Re-connecting Computer Sites (discussed in this section)
  2. UVa 10034 - Freckles (straightforward MST problem)
  3. **UVa 11228 - Transportation System *** (split output for short versus long edges)
  4. **UVa 11631 - Dark Roads *** (weight of all graph edges - weight of all MST edges)
  5. UVa 11710 - Expensive Subway (output 'Impossible' if graph still unconnected)
  6. UVa 11733 - Airports (maintain cost at every update)
  7. **UVa 11747 - Heavy Cycle Edges *** (sum the edge weights of the chords)
  8. UVa 11857 - Driving Range (find weight of the last edge added to MST)
  9. LA 3171 - Oreon (Manila06, MST)
  10. LA 4138 - Anti Brute Force Lock (Jakarta08, the underlying problem is MST)
  11. 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 10048 - Audiophobia *** (minimax, also solvable with Floyd Warshall's)
  4. UVa 10099 - Tourist Guide (maximin, also solvable with Floyd Warshall's)
  5. UVa 10147 - Highways (partial 'minimum' spanning tree)
  6. **UVa 10369 - Arctic Networks *** (minimum spanning 'forest')
  7. UVa 10397 - Connect the Campus (partial 'minimum' spanning tree)
  8. UVa 10462 - Is There A Second Way Left? (second best spanning tree)
  9. **UVa 10600 - ACM Contest and Blackout *** (second best spanning tree)
  10. UVa 10842 - Traffic Flow (find minimum weighted edge in 'maximum' spanning tree)
  11. LA 3678 - The Bug Sensor Problem (Kaohsiung06, minimum spanning 'forest')
  12. LA 4110 - RACING (Singapore07, 'maximum' spanning tree)
  13. LA 4848 - Tour Belt (Daejeon10, modified MST)

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 the other inventor of MST algorithm: 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.

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 **Tarjan's Strongly Connected Components algorithm** (discussed in this section together with other DFS variants invented by him and his colleagues). 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**.

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.

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*[8] *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 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 weight = 1), 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 approaches are discussed below.

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 4.2) turns BFS into a natural choice for Single-Source Shortest Paths (SSSP) problems on *unweighted* graphs. This is because in unweighted graph, the distance between two neighboring vertices connected with an edge is simply one unit. Thus the layer count of a vertex that we have seen previously is precisely the shortest path length from the source to that vertex. For example in Figure 4.2, the shortest path from the vertex labeled with '35' to the vertex labeled '30', is 3, as '30' is in the third layer in BFS sequence of visitation.

Some programming problems require us to reconstruct the actual shortest path, not just the shortest path length. For example, in Figure 4.2, above, the shortest path from '35' to '30' is $35 \rightarrow 15 \rightarrow 10 \rightarrow 30$. Such reconstruction is easy if we store the shortest path (actually BFS) spanning tree[9]. 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 '30' remembers '10' as its parent, vertex '10' remembers '15', vertex '15' remembers '35' (the source). To reconstruct the actual shortest path, we can do a simple recursive backtracking from the last vertex '35' until we hit the source vertex '30'. The modified BFS code (check the comments) is relatively simple:

```
// inside int main()
  map<int, int> dist; dist[s] = 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.count(v.first)) {
        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 target vertex t
```

[8]This generic SSSP problem can also be used to solve: 1). Single-Pair 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.

[9]Reconstructing the shortest path is not shown in the next two subsections (Dijkstra's and Bellman Ford's) but the idea is the same as the one shown here.

```
void printPath(int u) {    // simple function to extract information from 'vi p'
  if (u == s) { printf("%d", s); return; }         // base case, at the source s
  printPath(p[u]);   // recursive call: to make the output format: s -> ... -> t
  printf(" %d", u); }
```

Example codes: ch4_04_bfs.cpp; ch4_04_bfs.java

We would like to remark that recent competitive programming problems involving BFS are no longer written as straightforward SSSP problems, but 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), BFS with single destination – solved by reversing the role of source and destination (UVa 11513), BFS with non-trivial states (UVa 321, 10150), 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 also run BFS from more than one source. 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?

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/edges involved in the path) but has smaller total weight than the 'shorter' path found by BFS. For example, in Figure 4.16, the shortest path from source vertex 2 to vertex 3 is not via direct edge $2 \rightarrow 3$ with weight 7 that is normally found by BFS, but a 'detour' path: $2 \rightarrow 1 \rightarrow 3$ with smaller weight $2 + 3 = 5$.

To solve SSSP problem on weighted graph, we use a *greedy* Edsger Wybe *Dijkstra*'s algorithm. There are several ways to implement this classic algorithm. 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 amount of code *minimal* – a necessary feature in competitive programming.

This Dijkstra's variant maintains a **priority** queue called `pq` that stores pairs of vertex information. The first item of the pair is the distance of that vertex from source and the second item is the vertex number. This `pq` is sorted based on increasing distance from source, and if tie, by vertex number. Unlike the original version of Dijkstra's algorithm (see [3]) that uses binary heap feature that is not supported in built-in library[10], `pq` only contains one item initially, the base case: `(0,s)`. Then, this Dijkstra's variant repeat the following process until `pq` is empty. It greedily takes out vertex information pair (d, u) from the front of `pq`. If the distance of u from source recorded in the first item of the pair equals to `dist[u]`, it process u; otherwise, it ignores u. The reason for this check is shown below. When the algorithm process u, it tries to relax[11] all neighbor v of u. Every time it is able to relax an edge $u-v$, it will *enqueue* pair (new distance to v from source, v) into `pq`. This flexibility allows *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 one with the correct (smaller) distance. The code is shown below and it looks very similar to BFS code shown in Section 4.2.2.

Exercise 4.4.3.1: This variant may be different from what you learn from other books (e.g. [3, 23, 4]). Analyze if this variant still runs in $O((V + E) \log V)$?

Exercise 4.4.3.2: 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 has minimal coding. It uses `set`. Implement this variant!

[10]The original version of Dijkstra's requires `heapDecreaseKey` operation in binary heap DS that is not supported by built-in priority queue. This Dijkstra's variant uses only two basic priority queue operations: enqueue and dequeue.

[11]The operation: `relax(u, v, w_u_v)` sets `dist[v] = min(dist[v], dist[u] + w_u_v)`.

```
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));
                              // ^to sort the pairs by increasing distance from s
while (!pq.empty()) {                                                  // main loop
  ii front = pq.top(); pq.pop();      // greedy: pick shortest unvisited vertex
  int d = front.first, u = front.second;
  if (d == dist[u])              // this check is important, see the explanation
    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));
}   } }  // note: this variant can cause duplicate items in the priority queue
```

Example codes: ch4_05_dijkstra.cpp; ch4_05_dijkstra.java

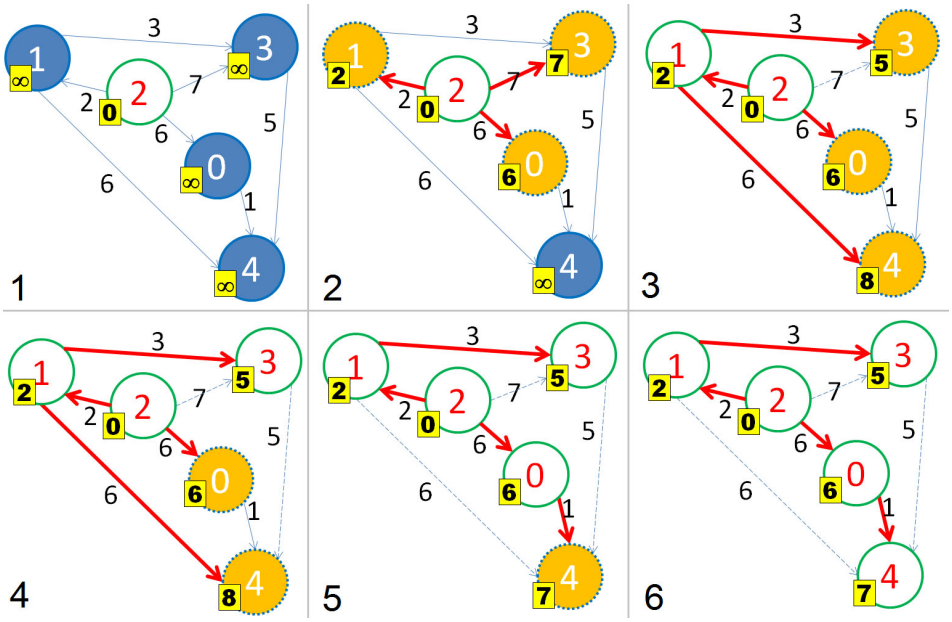


Figure 4.16: Dijkstra Animation on a Weighted Graph (from UVa 341 [28])

Figure 4.16 shows an example of running this Dijkstra's implementation variant on a small graph.

1. At the beginning, only `dist[source] = 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. But it is OK, as our Dijkstra's variant will only pick the one with minimal distance later.
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 `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)}, 2 entries of vertex 4.
6. Now, (7,3) can be ignored as we know that its `d > dist[3]` (i.e. 7 > 5). Then (7,4) is processed as before but nothing change. And finally (8,4) is ignored again as its `d > dist[4]` (i.e. 8 > 7). This Dijkstra's variant stops here as the pq is empty.

4.4.4 SSSP on Graph with Negative Weight Cycle

If the input graph has negative edge weight, the *original* Dijkstra's algorithm can produce wrong answer. However, the Dijkstra's implementation variant shown above will work just fine, albeit being slower. Try running the variant on the graph in Figure 4.17.

This is because the variant will keep inserting new vertex information pair to 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 source have been found). However, when given a graph with negative weight *cycle*, the variant – if implemented as shown above – will be trapped in an infinite loop.

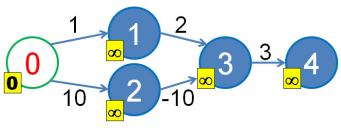


Figure 4.17: -ve Weight

Example: See the graph in Figure 4.18. Path 1-2-1 is a negative cycle. The weight of this cycle is 15 + (-42) = -27.

To solve the SSSP problem in the presence of negative weight *cycle*, 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 V-1 times!

Initially `dist[s] = 0`, the base case. If we relax an edge $s - u$, then `dist[u]` will have the correct value. If we then relax an edge $u - v$, 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 the code is simpler than BFS and Dijsktra's implementation:

```
vi dist(V, INF); dist[s] = 0;
for (int i = 0; i < V - 1; i++)  // relax all E edges V-1 times, overall O(VE)
  for (int u = 0; u < V; u++)                          // these two loops = O(E)
    for (int j = 0; j < (int)AdjList[u].size(); j++) {
      ii v = AdjList[u][j];        // we can record SP spanning here if needed
      dist[v.first] = min(dist[v.first], dist[u] + v.second);          // relax
    }
```

Example codes: ch4_06_bellman_ford.cpp; ch4_06_bellman_ford.java

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 (much) slower compared to Dijkstra's. However, the way Bellman Ford's works ensure that it will never be trapped in 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 although such SSSP problem is ill-defined. After relaxing all E edges V-1 times, the SSSP problem should have been solved, i.e. we cannot relax any more edge. If there is a negative cycle, we can still relax an edge! Example: In Figure 4.18, 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] = 988` and `dist[1] = 946` (right). As there is a negative cycle, we can still do this again, i.e. relaxing `dist[1] = 946+15 = 961` < 988!

In programming contests, these properties of Bellman Ford's cause it to be used only to solve the SSSP problem on *small* graph which is *not guaranteed* to be free from negative weight cycle.

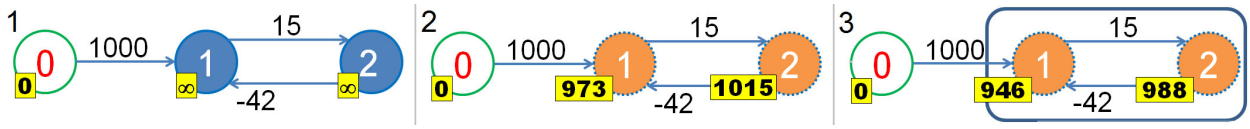


Figure 4.18: Bellman Ford's can detect the presence of negative cycle (from UVa 558 [28])

Exercise 4.4.4.1: A known improvement for Bellman Ford's (especially among Chinese programmers) is the SPFA (Shortest Path Faster Algorithm). Investigate this variant in Internet!

```
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");
```

Programming Exercises related to Single-Source Shortest Paths:

- On Unweighted Graph: BFS
  1. UVa 00314 - Robot (state: (position, direction), transform input graph a bit)
  2. **UVa 00321 - The New Villa *** (state: (position, bitmask 2^{10}), print the path)
  3. UVa 00336 - A Node Too Far (discussed in this section)
  4. UVa 00383 - Shipping Routes (simple SSSP solvable with BFS)
  5. UVa 00429 - Word Transformation (connect 2 words with an edge if differ by 1 letter)
  6. UVa 00439 - Knight Moves (the edges are defined by chess knight rule)
  7. UVa 00532 - Dungeon Master (3-D BFS)
  8. UVa 00567 - Risk (simple SSSP solvable with BFS)
  9. UVa 00627 - The Net (print the path)
  10. UVa 00762 - We Ship Cheap (simple SSSP solvable with BFS)
  11. UVa 00924 - Spreading the News (the spread is like BFS traversal)
  12. UVa 00928 - Eternal Truths (state: ((direction, step), (row, col)))
  13. UVa 10009 - All Roads Lead Where? (simple SSSP solvable with BFS)
  14. UVa 10044 - Erdos numbers (the parsing part is troublesome)
  15. **UVa 10047 - The Monocyle *** (state: ((row, col), (direction, color)); BFS)
  16. UVa 10067 - Playing with Wheels (implicit graph in problem statement)
  17. UVa 10150 - Doublets (BFS state is string!)
  18. UVa 10422 - Knights in FEN (solvable with BFS)
  19. UVa 10610 - Gopher and Hawks (solvable with BFS)
  20. UVa 10653 - Bombs; NO they are Mines (BFS implementation must be efficient)
  21. UVa 10959 - The Party, Part I (SSSP from source 0 to the rest)
  22. UVa 10977 - Enchanted Forest (BFS with blocked states)
  23. **UVa 11101 - Mall Mania *** (multi-sources BFS fr m1, get min at border of m2)
  24. UVa 11049 - Basic Wall Maze (some restricted moves, print the path)
  25. UVa 11352 - Crazy King (filter the graph first, then it becomes SSSP)
  26. UVa 11513 - 9 Puzzle (reverse the role of source and destination)
  27. UVa 11624 - Fire (multi-sources BFS)
  28. UVa 11792 - Krochanska is Here (be careful with the definition of 'important station')
  29. UVa 11974 - Switch The Lights (BFS on implicit unweighted graph)
  30. LA 4408 - Unlock the Lock (KualaLumpur08, SSSP solvable with BFS)
  31. LA 4637 - Repeated Substitution ... (Tokyo09, SSSP solvable with BFS)
  32. LA 4645 - Infected Land (Tokyo09, SSSP solvable with BFS)
- On Weighted Graph: Dijkstra's
  1. UVa 00341 - Non-Stop Travel (actually solvable with Floyd Warshall's algorithm)
  2. UVa 00929 - Number Maze (on a 2D maze/implicit graph)
  3. UVa 10166 - Travel (this can be modeled as a shortest paths problem)
  4. UVa 10269 - Adventure of Super Mario (solvable with Dijkstra's)
  5. UVa 10278 - Fire Station (Dijkstra's from fire stations to all intersections + pruning)
  6. UVa 10389 - Subway (use geometry skill to build weighted graph, then run Dijkstra's)

7. UVa 10603 - Fill (state: (a, b, c), source: (0, 0, c), 6 possible transitions)
8. **UVa 10801 - Lift Hopping *** (model the graph carefully!)
9. UVa 10986 - Sending email (straightforward Dijkstra's application)
10. **UVa 11367 - Full Tank? *** (model the graph carefully[12])
11. UVa 11377 - Airport Setup (model the graph carefully[13])
12. **UVa 11492 - Babel *** (model the graph carefully!)
13. UVa 11833 - Route Change (stop Dijkstra's at service route path + some modification)
14. LA 3133 - Finding Nemo (Beijing04, SSSP, Dijkstra's on grid)
15. LA 4204 - Chemical Plant (Dhaka08, build graph first, SSSP, Dijkstra's)
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 10557 - XYZZY *** (check 'positive' cycle, check connectedness!)
  3. **UVa 11280 - Flying to Fredericton *** (modified Bellman Ford's)

Profile of Algorithm Inventors

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**. He does not like 'GOTO' statement and influenced the widespread deprecation of 'GOTO' and its replacement: structured control constructs. His famous 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 Delbert Ray Fulkerson on the max flow problem and the **Ford Fulkerson's algorithm** for solving it, established the max-flow min-cut theorem.

Robert W Floyd (1936-2001) was an eminent American computer scientist. Floyd's contributions include the design of **Floyd's algorithm**, 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. He is the inventor of the **transitive closure algorithm**, now known as **Warshall's algorithm**. This algorithm was later named as Floyd Warshall's because Robert W Floyd independently invented essentially similar algorithm.

Delbert Ray Fulkerson (1924-1976) was a mathematician who co-developed the **Ford Fulkerson's method**, one of the most used algorithms to solve the maximum flow problem in networks. In 1956, he published his paper on the Ford Fulkerson's method together with Lester R Ford.

Jack R. Edmonds (born 1934) is a mathematician. He and Richard Karp invented the **Edmonds Karp's algorithm** for computing the maximum flow in a flow network in $O(VE^2)$. 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**. However, his most important contribution is probably the Edmonds's **matching/blossom shrinking algorithm** – one of the most cited papers.

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 maximum flow problem.

[12] State: (location, fuel), source s = (s, 0), sink t = (e, any), only enqueue fuel + 1.

[13] 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.

4.5 All-Pairs Shortest Paths

4.5.1 Overview and Motivation

Motivating Problem: Given a connected, weighted graph G with $V \leq 200$ and two vertices $s1$ and $s2$, find a vertex v in G that represents the best meeting point, i.e. `dist[s1][v] + dist[s2][v]` is the minimum over all possible v. What is the best solution?

This problem requires the shortest path information from two sources s1 and s2 to all vertices in G. This can be easily done with two calls of Dijkstra's algorithm that we have learned earlier. One from s1 to produce shortest distance array `dist1` from s1, and one from s2 to produce `dist2`. Then iterates through all possible vertices in graph to find v such that `dist1[v] + dist2[v]` is minimized. However, can we solve this problem in a *shorter way*?

If the given graph is known to have $V \leq 200$, then there is an even 'simpler' algorithm – in terms of implementation – to solve this problem *as quickly as possible*!

Load the small graph into an Adjacency Matrix and then run the following short 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 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

  for (int k = 0; k < V; k++) // common error: 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]);
```

Example codes: `ch4_07_floyd_warshall.cpp`; `ch4_07_floyd_warshall.java`

Exercise 4.5.1.1: Is there any specific reason why `AdjMat[i][j]` must be set to 1B (1000000000) 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 make a differentiation between graph with negative weight edges but no negative cycle and graph with negative cycle. Will this short Floyd Warshall's algorithm works on graph on negative weight and/or negative cycle? Do some experiment!

This algorithm is called Floyd Warshall's algorithm, invented by Robert W *Floyd* and Stephen *Warshall*. Floyd Warshall's is a DP algorithm that clearly runs in $O(V^3)$ due to its 3 nested loops[14], but since $|V| \leq 200$ for the given problem, this is do-able. 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 algorithms 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 – 4 short lines only. If the given graph is small ($V \leq 200$), do not hesitate to use this algorithm – even if you only need a solution for the SSSP problem.

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 and must be solved with DP techniques (see Section 4.7.1).

[14]Floyd Warshall's algorithm must use Adjacency Matrix so that the weight of edge(i, j) can be accessed in $O(1)$.

The basic idea behind Floyd Warshall's is to gradually allow the usage of intermediate vertices to form the shortest paths. Let the vertices be labeled from 0 to 'V-1'. We start with direct edges only, i.e. shortest path of vertex i to vertex j, denoted as `sp(i,j,-1)` = weight of edge (i,j). Then, find shortest paths between any two vertices with the help of restricted intermediate vertices from vertex `[0..k]`. First, we only allow $k = 0$, then $k = 1$, $k = 2 \ldots$, up to $k = V$-1.

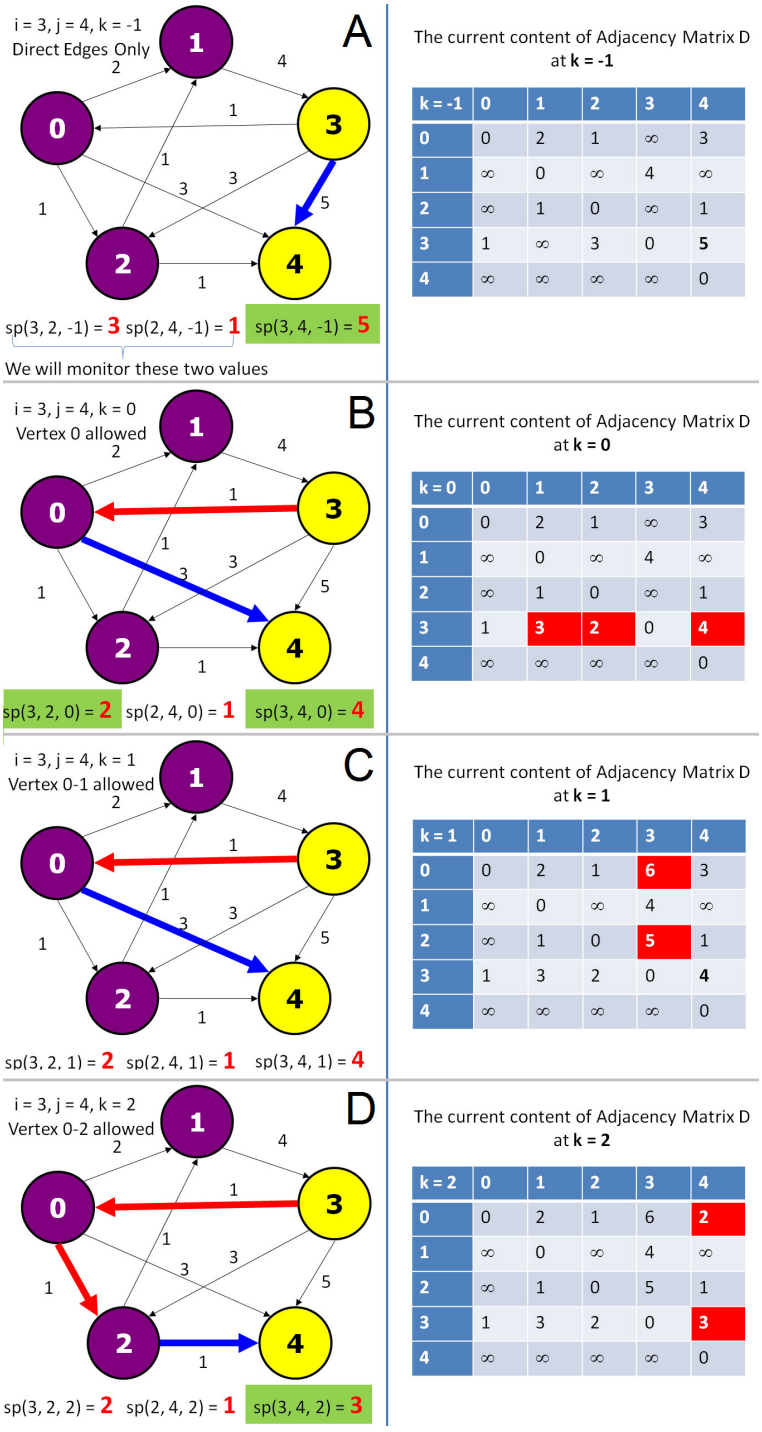
In Figure 4.19, we want to find `sp(3,4,4)`. The shortest path is 3-0-2-4 with cost 3. But how to reach this solution? We know that by using only direct edges, `sp(3,4,-1) = 5`, as in Figure 4.19.A. The solution for `sp(3,4,4)` will *eventually* be reached from `sp(3,2,2)+sp(2,4,2)`. But with using only direct edges, `sp(3,2,-1)+sp(2,4,-1) = 3+1 = 4` is still > 3.

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 in Figure 4.19.B. 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 pairs considering only vertex 0 as the intermediate vertex.

When we allow $k = 1$, i.e. vertex 0 and 1 can now be used as the intermediate vertices, then it happens that there is no change to `sp(3,4,1)`, `sp(3,2,1)`, nor to `sp(2,4,1)`.

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`.

Floyd Warshall's repeats this process for $k = 3$ and finally $k = 4$ but `sp(3,4,4)` remains at 3.



k = -1	0	1	2	3	4
0	0	2	1	∞	3
1	∞	0	∞	4	∞
2	∞	1	0	∞	1
3	1	∞	3	0	5
4	∞	∞	∞	∞	0

k = 0	0	1	2	3	4
0	0	2	1	∞	3
1	∞	0	∞	4	∞
2	∞	1	0	∞	1
3	1	3	2	0	4
4	∞	∞	∞	∞	0

k = 1	0	1	2	3	4
0	0	2	1	6	3
1	∞	0	∞	4	∞
2	∞	1	0	5	1
3	1	3	2	0	4
4	∞	∞	∞	∞	0

k = 2	0	1	2	3	4
0	0	2	1	6	2
1	∞	0	∞	4	∞
2	∞	1	0	5	1
3	1	3	2	0	3
4	∞	∞	∞	∞	0

Figure 4.19: Floyd Warshall's Explanation

We define $D^k_{i,j}$ to 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:
$D^{-1}_{i,j} = weight(i, j)$. This is the base case when we do not use any intermediate vertices.
$D^k_{i,j} = min(D^{k-1}_{i,j}, D^{k-1}_{i,k} + D^{k-1}_{k,j}) = min(\text{not using vertex } k, \text{using } k)$, 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 $D^2_{3,4}$, (row 3, column 4, in table $k = 2$, index start from 0), we look at the minimum of $D^1_{3,4}$ or the sum of $D^1_{3,2} + D^1_{2,4}$ (see Table 4.2). The naïve implementation is to use a 3-dimensional matrix `D[k][i][j]` of size $O(V^3)$. However, as 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'. Thus, the Floyd Warshall's algorithm just need $O(V^2)$ space although it still runs in $O(V^3)$.

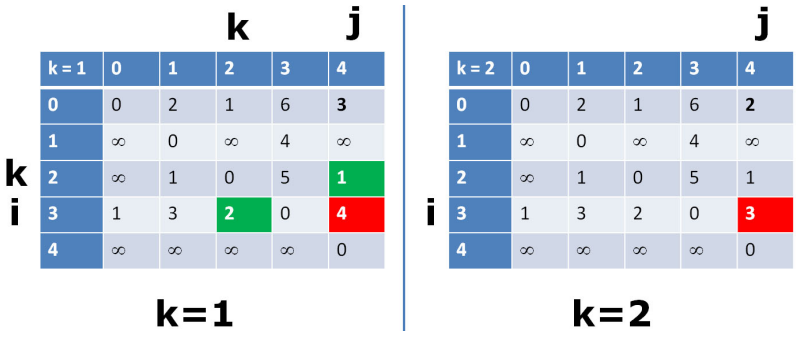


Table 4.2: Floyd Warshall's DP Table

4.5.3 Other Applications

The basic purpose of Floyd Warshall's algorithm is to solve the APSP problem. Here we list down several problem variants that are also solvable with Floyd Warshall's.

Printing Shortest Paths

A common issue encountered by programmers who use the 4 liners Floyd Warshall's without understanding how it works is when they are asked to print the shortest paths too. Previously, 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. For Floyd Warshall's algorithm, we need to store a 2D parent matrix. The modified code is shown below.

```
// inside int main()
  // let p be the 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* developed an algorithm for the Transitive Closure problem: Given a graph, determine if vertex i is connected to j. This variant uses logical bitwise operators which is much faster than arithmetic operators. Initially, `d[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 `d[i][j]`.

```
for (int k = 0; k < V; k++)
  for (int i = 0; i < V; i++)
    for (int j = 0; j < V; j++)
      d[i][j] = d[i][j] | (d[i][k] & d[k][j]);
```

Exercise 4.5.3.1: How to solve transitive closure problem for a *directed* graph with $V \leq 1000$?

Minimax and Maximin (Revisited)

We have seen the minimax path problem earlier in Section 4.3.4. The solution using Floyd Warshall's is shown below. First, initialize `d[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 `d[i][j] = INF`. Then, we try all possible intermediate vertex k. The minimax cost `d[i][j]` is the minimum of either (itself) or (the maximum between `d[i][k]` or `d[k][j]`).

```
for (int k = 0; k < V; k++)
  for (int i = 0; i < V; i++)
    for (int j = 0; j < V; j++)
      d[i][j] = min(d[i][j], max(d[i][k], d[k][j]));
```

Exercise 4.5.3.2: Solve the maximin path problem using Floyd Warshall's!

Finding Negative Cycle

In Section 4.4.4, we have seen how Bellman Ford's algorithm – since it relax all edges at most V-1 times – can be used to check if the given graph has negative cycle(s). Floyd Warshall's can also be used to detect if the given graph has negative cycle(s). If the given graph indeed has negative cycle(s), then after running the $O(V^3)$ Floyd Warshall's on the Adjacency Matrix of the graph, the *main diagonal* of the Adjacency Matrix contains *at least one negative value*. If `AdjMat[i][i] < 0`, this means that the 'shortest' (actually cyclic) path from a vertex i that goes through up to V-1 other intermediate vertices and returns back to i has negative value. Repeating the 'shortest' path one more time will give an even smaller 'shortest' path which implies a negative cycle.

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): if 1.0 United States dollar (USD) buys 0.5 British pounds (GBP), 1.0 GBP buys 10.0 French francs (FRF), 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 *negative cycle*. Solve the arbitrage problem using Floyd Warshall's!

Remarks About SSSP/APSP algorithms for Programming Contests

All three algorithms: Dijkstra's/Bellman Ford's/Floyd Warshall's are used to solve the general case of SSSP/APSP on weighted graph. Out of these three, Bellman Ford's is rarely used in practice due to its high time complexity. It is only useful if the problem setter gives a 'reasonable size' graph with negative cycle. For general cases, Dijkstra's is the best solution for the SSSP problem for 'reasonable size' weighted graph without negative cycle. However, when the given graph is small

$V \leq 200$ – which happens many times, it is clear that Floyd Warshall's is the best way to go. You can see that by looking at the list of UVa problems in this book. There are more problems solvable with Floyd Warshall's than Dijkstra's or Bellman Ford's.

Perhaps the reason is because sometimes the problem setter includes shortest paths as *sub-problem* of the original, (much) more complex, problem. To make the problem still manageable during contest time, he purposely sets the input size to be small so that the shortest paths sub-problem is solvable with 4 liner Floyd Warshall's (UVa 10171, 10793, 11463)

Today's trend related to (pure) shortest paths problem involves careful *graph modeling* (UVa 10801, 11367, 11492). To do well in programming contests, make sure that you have this soft skill: the ability to spot the graph in the problem statement.

In Section 4.7.1, we will revisit some shortest paths problem on Directed Acyclic Graph (DAG). This important variant is solvable with generic Dynamic Programming (DP) technique. We will also present another way of viewing DP technique as 'traversal on DAG' in that section.

In Table 4.3, we present an SSSP/APSP algorithm decision table within the context of programming contest. This is to help the readers in deciding which algorithm to choose depending on various graph criteria where 'Best' is the most suitable algorithm, 'Ok' is a correct algorithm but not the best, 'Bad' is a (very) slow algorithm, and 'WA' is an incorrect algorithm.

Graph Criteria	BFS $O(V+E)$	Dijkstra's variant $O((V+E)\log V)$	Bellman Ford's $O(VE)$	Floyd Warshall's $O(V^3)$
Max Graph Size	$V, E \leq 1M$	$V, E \leq 50K$	$VE \leq 1M$	$V \leq 200$
Unweighted	Best	Ok	Bad	Bad
Weighted	WA	Best	Ok	Bad
Negative weight	WA	Our variant Ok	Ok	Bad
Negative cycle	Cannot detect	Cannot detect	Can detect	Can detect
Small graph	WA if weighted	Overkill	Overkill	Best

Table 4.3: 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 00186 - Trip Routing (graph is small)
  2. UVa 00423 - MPI Maelstrom (graph is small)
  3. **UVa 00821 - Page Hopping *** (one of the 'easiest' ICPC World Finals problem)
  4. **UVa 10171 - Meeting Prof. Miguel *** (solution is easy with APSP information)
  5. UVa 10724 - Road Construction (adding one edge will only change 'few things')
  6. UVa 10793 - The Orc Attack (Floyd Warshall's simplifies this problem)
  7. UVa 10803 - Thunder Mountain (graph is small)
  8. UVa 10947 - Bear with me, again.. (graph is small)
  9. UVa 11015 - 05-32 Rendezvous (graph is small)
  10. **UVa 11463 - Commandos *** (solution is easy with APSP information)
  11. LA 2818 - Geodetic Set Problem (Kaohsiung03, APSP, Floyd Warshall's)
  12. LA 4109 - USHER (Singapore07, SSSP, Floyd Warshall's, small graph)
  13. LA 4524 - Interstar Transport (Hsinchu09, APSP, Floyd Warshall's)
- Variants
  1. **UVa 00104 - Arbitrage *** (small arbitrage problem can be solved with FW)
  2. UVa 00125 - Numbering Paths (modified Floyd Warshall's)
  3. UVa 00436 - Arbitrage (II) (another arbitrage problem)
  4. **UVa 00334 - Identifying Concurrent ... *** (subproblem: transitive closure)
  5. UVa 00869 - Airline Comparison (run Warshall's 2x, then compare the AdjMatrices)
  6. **UVa 11047 - The Scrooge Co Problem *** (print path; if orig = dest, print twice)

4.6 Maximum Flow

4.6.1 Overview and Motivation

Motivating problem: Imagine a connected, weighted, and directed graph[15] as a pipe network where the pipes are the edges and the splitting points are the vertices. 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 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. See an illustration of Max Flow in Figure 4.20.

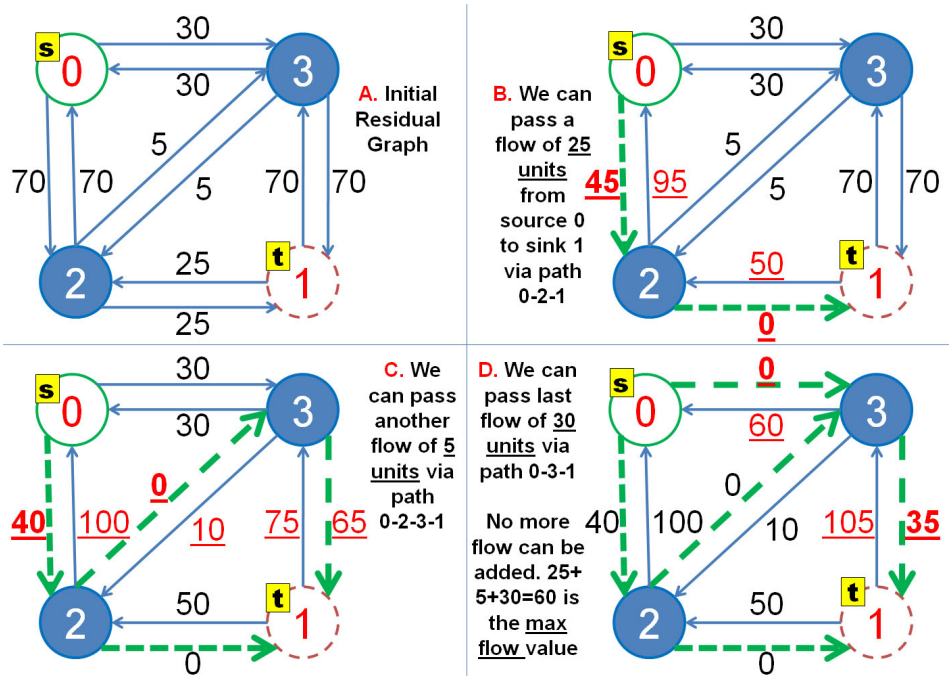


Figure 4.20: Illustration of Max Flow (From UVa 820 [28] - ICPC World Finals 2000 Problem E)

4.6.2 Ford Fulkerson's Method

One solution 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*. The pseudo code is like this:

```
setup directed residual graph; each edge has the same weight with original graph
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 positive 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 the capacity of forward edges (e.g. i -> j) along path p by f
       reason: obvious, we use the capacities of those forward edges
    3. increase the capacity of backward edges (e.g. j -> i) along path p by f
       reason: not so obvious, but this is important for the correctness of Ford
       Fulkerson's method; by increasing the capacity of a backward edge
       (j -> i), we allow later iteration/flow to cancel part of capacity used
       by a forward edge (i -> j) that is incorrectly used by earlier flow
  mf += f               // we can send a flow of size f from s to t, increase mf
}
output mf                                          // this is the max flow value
```

[15]Every weighted edge in an undirected graph can be transformed into two directed edges with the same weight.

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 4.21 where every path augmentation only decreases the (forward[16]) edge capacities along the path by 1. At the worst case, this is repeated $|f^*|$ times (it is 200 times in Figure 4.21). Because DFS runs in $O(E)$ in a flow graph[17], the overall time complexity is $O(|f^*|E)$ – we do not want this unpredictability in programming contests as the problem setter can choose to give (very) large $|f^*|$ value.

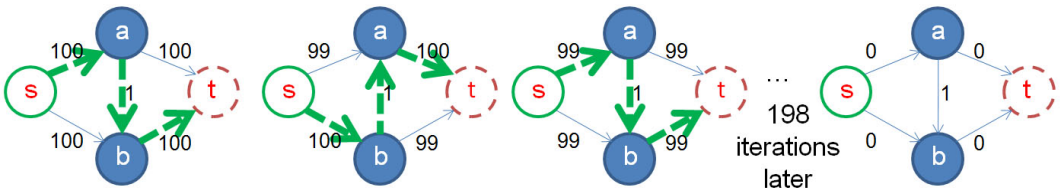


Figure 4.21: Ford Fulkerson's Method Implemented with DFS is Slow

4.6.3 Edmonds Karp's

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 is discovered by Jack *Edmonds* and Richard Manning *Karp*, thus named as Edmonds Karp's algorithm. 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 books like [3] 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 4.21: s→a→t (2 hops, send 100 unit flow) and s→b→t (2 hops, send another 100). That is, it does not 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 implementation that uses *only* Adjacency Matrix named as `res` with size $O(V^2)$ to store the residual[18] capacity of each edge. This version – which actually runs in $O(VE)$ BFS iterations × $O(V^2)$ per BFS due to Adjacency Matrix = $O(V^3E)$ – is fast enough to solve *some* (small-size) Max Flow problems listed in this section.

With this code, solving a Max Flow problem is now simpler. It is now a matter of: 1). Recognizing that the problem is indeed a Max Flow problem (this will get better after you solve more Max Flow problems); 2). Constructing the appropriate flow graph (i.e. if using our code below: Initiate the residual matrix `res` and set the appropriate values for 's' and 't').

Exercise 4.6.3.1: Before we continue, let's verify your understanding of Max Flow in Figure 4.22!

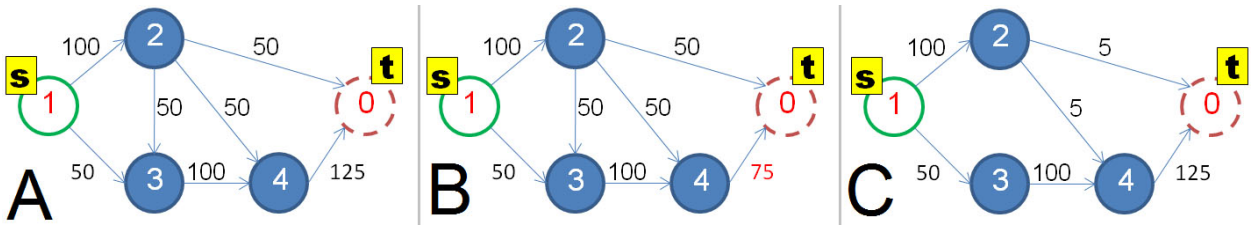


Figure 4.22: What are the Max Flow value of these three residual graphs?

[16]Note that after sending flow $s \rightarrow a \rightarrow b \rightarrow t$, the forward edge $a \rightarrow b$ is replaced by the backward edge $b \rightarrow a$, and so on. If this is not so, then the max flow value is just $1 + 99 + 99 = 199$ instead of 200 (wrong).

[17]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 $O(V + E)$.

[18]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 a flow pass through this edge with weight `f` ≤ `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`.

```
int res[MAX_V][MAX_V], mf, f, s, t;                          // global variables
vi p;    // p stores the BFS spanning tree from s, we use it to find path s -> t

void augment(int v, int minEdge) {     // traverse BFS spanning tree from s to t
  if (v == s) { f = minEdge; return; }  // record minEdge in a global variable f
  else if (p[v] != -1) { augment(p[v], min(minEdge, res[p[v]][v])); // recursive
                         res[p[v]][v] -= f; res[v][p[v]] += f; }       // update
}

// inside int main()
  // set up the 2d AdjMat 'res', 's', and 't' with appropriate values
  mf = 0;                                                  // mf stands for max_flow
  while (1) {              // O(VE^2) (actually O(V^3E) 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;     // three lines in one!
    }
    augment(t, INF); // find the minimum edge weight 'f' along 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
```

Example codes: ch4_08_edmonds_karp.cpp; ch4_08_edmonds_karp.java

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). We also do not need `vi dist` as `bitset` to flag whether a vertex has been visited or not is sufficient. Now modify the Edmonds Karp's implementation so that it really 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 residual capacity of each edge. A better way is to store the actual flow and capacity (not just the residual) of each edge as an $O(V+E)$ modified Adjacency List. We can derive the residual of an edge from the original capacity of that edge minus the flow of that edge. Now, modify the implementation again!

Exercise 4.6.3.4: Even the best Edmonds Karp's implementation still runs in $O(VE^2)$. Although it should be sufficient to solve most Max Flow problems in today's programming contests, some harder Max Flow problems need a faster solution. Explore the Internet to study and implement the $O(V^2E)$ Dinic's algorithm which is theoretically faster as $E >> V$ in flow graph.

Next, we show an example of modeling the flow (residual) graph of UVa 259 - Software Allocation[19]. The abridged version of this problem is as follows: You are given up to 26 app(lication)s (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

[19] 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.

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 '!'.

The flow graph formulation is as shown in Figure 4.23. First 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. Then, link apps to computers as mentioned in the problem description. Link source s to all apps and link all computers to sink t. We set all the edge weights to be 1 (this implies these edges can only be used once) *except* the edges from source 's' to apps (can be used several times). This is the key part of this flow graph.

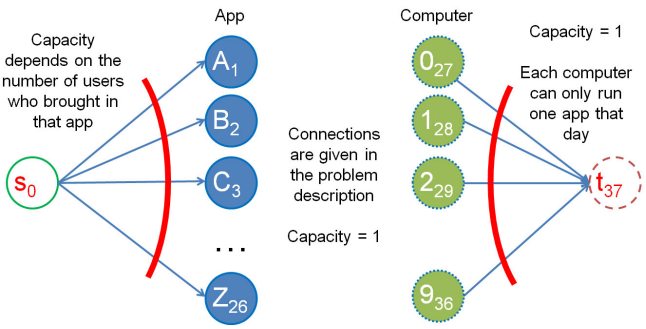


Figure 4.23: Residual Graph of UVa 259 [28]

The path from an app A to a computer B and finally to sink 't' have weight 1 that corresponds to *one matching* between that particular app A and computer B. The problem says that there can be *more than one* (say, X) users bringing in a particular app A in a given day. To accommodate this, we have to set the capacity from source 's' to this particular app A to X.

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 applicants 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' will have been found by the Edmonds Karp's algorithm.

The actual app → 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 → app) in the residual matrix `res` will contain a value +1 if the corresponding forward edge (app → computer) is selected in the paths that contribute to the Max Flow `mf`.

4.6.4 Other Applications

There are several other interesting applications/variants of the Max Flow problem. We discuss six examples here while some others are deferred until Section 4.7.4 (Bipartite Graph). 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-component, T-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, but we want to do so in the least cost way. As with bridges, Min Cut has applications in 'sabotaging' networks.

The solution is simple: The by-product of computing Max Flow is Min Cut! Let's see Figure 4.20.D. After Max Flow algorithm stops, we run graph traversal (DFS/BFS) from source s again. All vertices that are still reachable from source s (vertex 0 and 2 in this case) belong to the S-component. All other unreachable vertices belong to the T-component (vertex 1 and 3 in this case). All edges connecting S-component and 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 value is the minimum over all possible s-t cuts value.

Multi-source Multi-sink Max Flow

Sometimes, we can have more than one source and/or more than one sink. However, this variant is no harder than the original Max 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 the Edmonds Karp's algorithm as per normal. FYI, we have seen this variant as exercise in Section 4.2.2 (Multi-Sources BFS).

Max Flow with Vertex Capacities

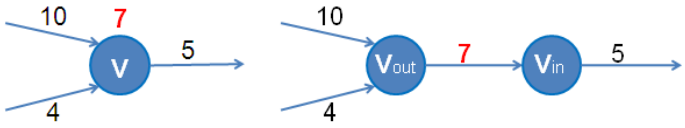


Figure 4.24: Vertex Splitting Technique

We can also have a Max Flow variant where the capacities are not just defined along the edges but *also on the vertices*. To solve this variant, we can use the vertex splitting technique. A weighted graph with a vertex weight can be converted into a more familiar one *without* a vertex weight by splitting each weighted vertex v to v_{out} and v_{in}, reassigning its incoming/outgoing edges to v_{out}/v_{in}, respectively and finally putting the original vertex v's weight as the weight of edge (v_{out}, v_{in}). See Figure 4.24 for illustration. Then run the Edmonds Karp's algorithm as per normal.

Maximum Independent Paths

The problem of finding the maximum number of independent paths from source s to sink t can be reduced to the Max Flow problem. Two paths are said to be independent if they do not share any vertex apart from s and t (vertex-disjoint). Solution: 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) 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.

Maximum 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 (i.e. two edge-disjoint paths can still share the same vertex). See Figure 4.25 for a comparison between independent paths and edge-disjoint paths from $s = 0$ to $t = 6$.

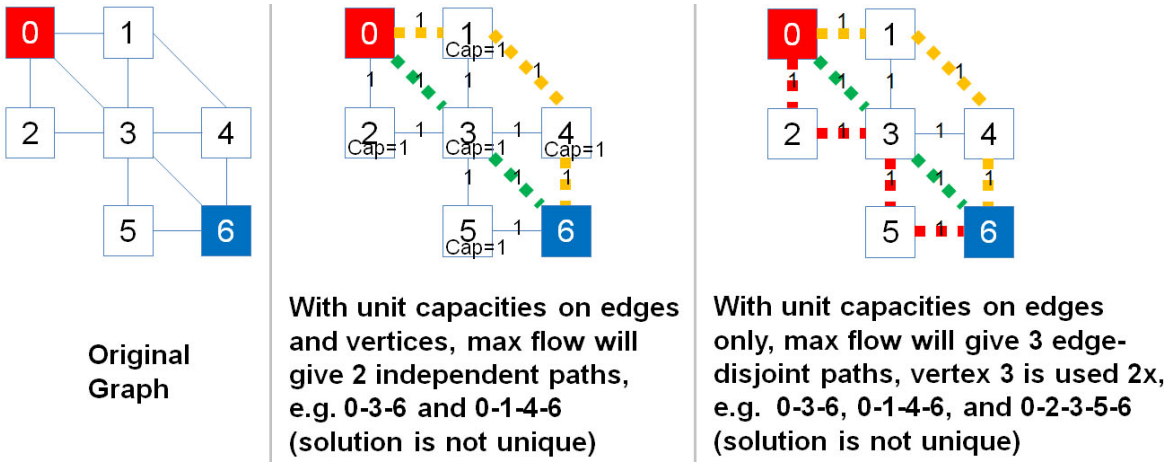


Figure 4.25: Comparison Between the Max Independent Paths versus Max Edge-Disjoint Paths

Min Cost (Max) Flow

The Min Cost Flow problem is the problem of finding the *cheapest* possible way of sending a certain amount of flow (usually the Max Flow) through a flow network. In this problem, every edge has two attributes: the flow capacity through this edge *and the cost* to use this edge.

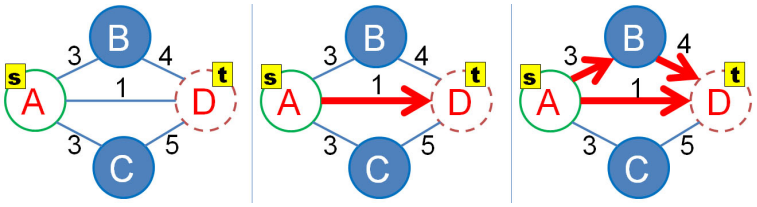


Figure 4.26: An Example of Min Cost Max Flow (MCMF) Problem (from UVa 10594 [28])

Figure 4.26, left shows a (modified) instance of UVa 10594. Here, each edge has a uniform capacity of 10 units and a cost as shown in the edge label. We want to send 20 units of flow which can be satisfied by sending 10 units of flow $A \rightarrow D$ with cost $1 \times 10 = 10$ (Figure 4.26, middle); plus another 10 units of flow $A \rightarrow B \rightarrow D$ with cost $(3 + 4) \times 10 = 70$ (Figure 4.26, 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 (another 10 units), 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.

The Min Cost (Max) Flow, or in short MCMF, can be solved by replacing BFS to find the shortest (in terms of number of hops) augmenting path in Edmonds Karp's to Bellman Ford's to find the shortest/cheapest (in terms of the path cost) augmenting path. The Dijkstra's implementation variant shown in Section 4.4.3 can also be used as it can also handle negative edge weight that may appear when we cancel certain flow along backward edge.

Exercise 4.6.4.1: We have not shown any MCMF code in this section but instead we left the task of modifying the given Edmonds Karp's implementation (which uses BFS with residual matrix `res`) to a working MCMF implementation (which uses Bellman Ford's/Dijkstra's implementation variant with both `cap` and `flow` matrices where `res = cap - flow`) as exercise. Please write the modification by following these steps and then test your implementation on UVa 10594 (pure MCMF problem) and 10746 (minimum *weighted* bipartite matching):

a. Replace BFS code with Bellman Ford's code,
b. In finding augmenting path, instead of checking `res[u][v] > 0`, use `flow[u][v] < cap[u][v]`,
c. In finding cheapest augmenting path, be careful with the status of an edge(u, v):
...c.1. If it is a forward flow, *add the cost* of edge, *add flow* along that edge
...c.2. If it is a backward flow, *subtract*[20] *the cost* of that edge, *reduce flow* along that edge

Remarks About Edmonds Karp's for Programming Contests

When a 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 Max Flow in a glance. The hardest part for the contestant is to realize that the underlying problem is indeed a Max Flow problem and model the flow graph correctly.

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 his effort in optimizing his Max Flow code (perhaps a good Dinic's algorithm implementation which is not yet discussed in the current version of this book) and attempting various Max Flow problems available in many online judges to increase his aptitude towards Max Flow problems and its variants.

In IOI, Max Flow (and its variants) is currently outside the 2009 syllabus [10]. So IOI contestants can skip this section without penalty. However, it may be a good idea to learn these material 'ahead of time' to improve your skills with graph problems.

[20] 'Canceled flow' is the reason that the flow graph in MCMF can have negative weight edge.

Programming Exercises related to Max Flow:

- Standard Max Flow/Min Cut (Edmonds Karp's)
  1. **UVa 00259 - Software Allocation *** (discussed in this section)
  2. UVa 00753 - A Plug for Unix (matching, max flow, capacity between adapters is ∞)
  3. **UVa 00820 - Internet Bandwidth *** (max flow, discussed in this section)
  4. UVa 10092 - The Problem with the ... (MCBM variant, see Section 4.7.4)
  5. **UVa 10480 - Sabotage *** (min cut)
  6. UVa 10511 - Councilling (matching, max flow, print the assignment)
  7. UVa 10779 - Collectors Problem
- Variants
  1. UVa 00563 - Crimewave
  2. UVa 10330 - Power Transmission (max flow with vertex capacities)
  3. **UVa 10594 - Data Flow *** (basic min cost max flow problem)
  4. **UVa 10746 - Crime Wave - The Sequel *** (min *weighted* bip matching $\approx$ MCMF)
  5. UVa 10806 - Dijkstra, Dijkstra. (send 2 edge-disjoint flows with min cost)
  6. **UVa 11506 - Angry Programmer *** (min cut with vertex capacities)
  7. LA 4271 - Necklace (Hefei08, flow on graph)
  8. LA 4722 - Highway Patrol (Phuket09, MCMF)

4.7 Special Graphs

Some basic graph problems have simpler/faster polynomial algorithms if the given graph is *special*. So far we have identified the following special graphs that commonly appear in programming contests: **Directed Acyclic Graph (DAG)**, **Tree**, **Eulerian Graph**, and **Bipartite Graph**. Problem setters 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 [11]). In this section, we discuss some popular graph problems on these special graphs (see Figure 4.27) – many of which have been discussed earlier on general graphs. Note that bipartite graph is still excluded in the IOI syllabus [10].

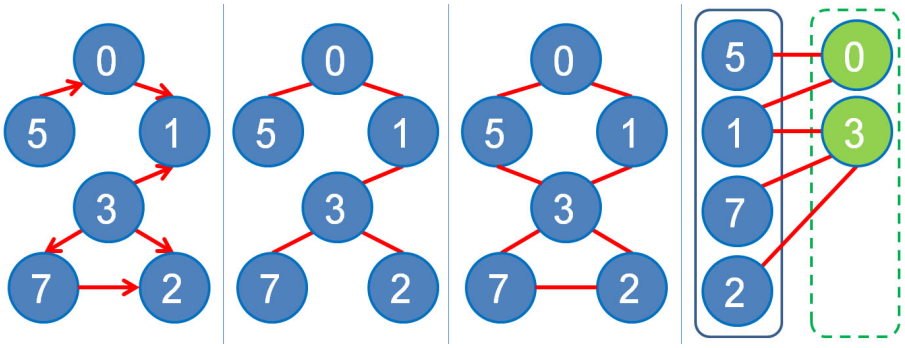


Figure 4.27: Special Graphs (L-to-R): DAG, Tree, Eulerian, Bipartite Graph

4.7.1 Directed Acyclic Graph

A Directed Acyclic Graph, abbreviated as DAG, is a special graph with the following characteristics: Directed and has no cycle. DAG guarantees the absence of cycle *by definition*. This makes DAG very suitable for Dynamic Programming (DP) techniques (see Section 3.5). After all, a DP recurrence must be *acyclic*. We can view DP states as vertices in 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 edges 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 distance value. This way, the distance values propagation is correct with just one $O(V+E)$ linear pass! This is the essence of Dynamic Programming principle to avoid recomputation of overlapping subproblem to covered earlier in Section 3.5.

The Single-Source *Longest Paths* problem is a problem of finding the longest (simple[21]) paths from a starting vertex s to other vertices. The decision[22] version of this problem is NP-complete on a general graph. 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[23]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.

Longest Paths on DAG has applications in project scheduling (see UVa 452). 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 (a.k.a. the *critical* path) found in this DAG. See Figure 4.28 for an example with 6 sub projects, their estimated completion time units, and their dependencies. The longest path $0 \rightarrow 1 \rightarrow 2 \rightarrow 4 \rightarrow 5$ 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.

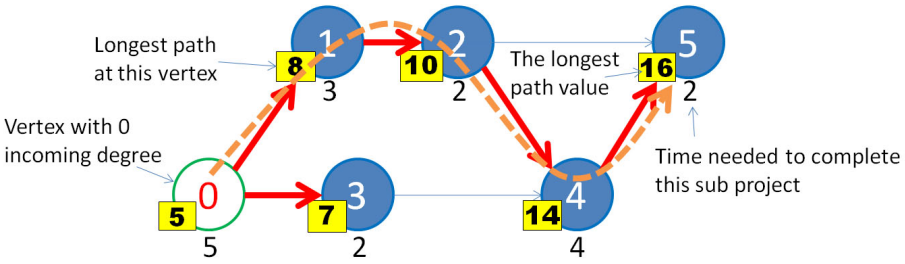


Figure 4.28: The Longest Path on this DAG is the Shortest Way to Complete the Project

Counting Paths in DAG

Motivating problem (UVa 988): 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).

Clearly 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 with a Dynamic Programming technique on DAG. First we compute one (any) topological order (vertex 0 will always be the first and the vertex that represents death/vertex n will always be the last). We start by setting `num_paths[0] = 1`. Then, we process the remaining vertices one by one according to the topological order. When processing vertex `cur`, we update each neighbor `v` of `cur` by setting `num_paths[v] += num_paths[cur]`. After such $O(V+E)$ steps, we will know the number of paths in `num_paths[n]`. Figure 4.29 shows an example with 9 events and 6 different possible life scenarios.

[21]On general graph with positive weight edges, longest path 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'. All paths in DAG are simple by definition.

[22]The decision version of this problem asks if the general graph has a simple path of total weight $\geq k$.

[23]The Longest Increasing Subsequence problem in Section 3.5.2 can also be modeled as Longest Paths on DAG.

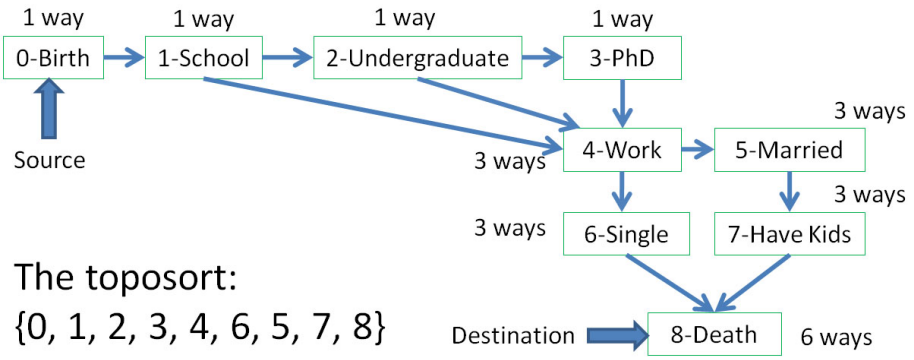


Figure 4.29: Example of Counting Paths in DAG

Exercise 4.7.1.1: A common variant of this problem is to count paths on a *grid* (an implicit graph) with restrictions that we can only go to right and down (thus making the movement acyclic). Solve the relevant UVa problems (UVa 825, 926, 11067, 11957) for practice!

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 an *implicit 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 101: 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 0 in such a way that the fishmonger has to pay as little tolls as possible to arrive at the market city $n-1$ 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 4.30, 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.

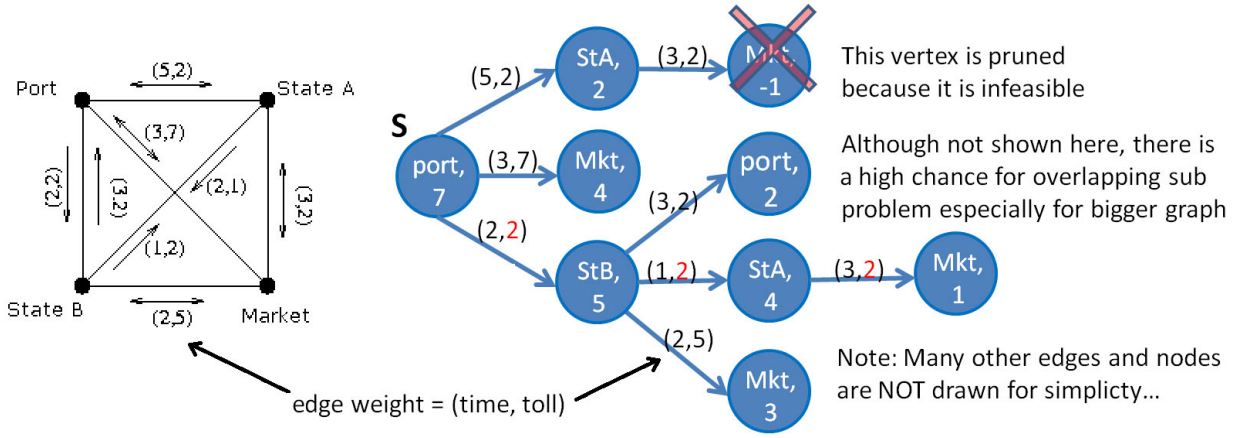


Figure 4.30: The Given General Graph (left) is Converted to DAG

It is clear after trying several test cases that the greedy Single-Source Shortest Paths (SSSP) algorithm like Dijkstra's (see Section 4.4.3) – on its pure form – does not work. Picking 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 $\leq t$. They are not independent!

However, if we attach a parameter: 't_left' to each vertex, then the given graph turns into a DAG as shown in Figure 4.30, right. We start with a modified vertex (port, t) in the DAG. Every time the fishmonger moves from a current city *cur* to another city X, it will move to a modified vertex (X, t - `travelTime[cur][X]`) in the DAG, with cost `toll[cur][X]`. Since time is a diminishing resource, we will never no longer encounter a cyclic situation. We can then use this (top down) Dynamic Programming recurrence: `go(cur, t_left)` to find the shortest path (in terms of total tolls paid) on this DAG. The detailed implementation is described with the following C++ code:

```
ii go(int cur, int t_left) {                          // top-down DP, returns a pair
  if (t_left < 0) return ii(INF, INF);                      // invalid state, prune
  if (cur == n - 1) return ii(0, 0);     // at market, toll = 0, time needed = 0
  if (memo[cur][t_left] != ii(-1, -1)) return memo[cur][t_left];    // revisited
  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 one with 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 and return the answer to the caller
```

There are only $O(nt)$ distinct states. Each state can be computed in $O(n)$. Overall time complexity is $O(n^2t)$ – do-able. The answer is in `go(0, t)`.

2. Minimum Vertex Cover (on a Tree)

Tree data structure is also an acyclic data structure. But unlike DAG, there is no overlapping subtrees in a tree. Thus, a standard tree is actually not suitable for Dynamic Programming (DP) technique. 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[24]) named as the 'DP on Tree' problems in competitive programming terminology.

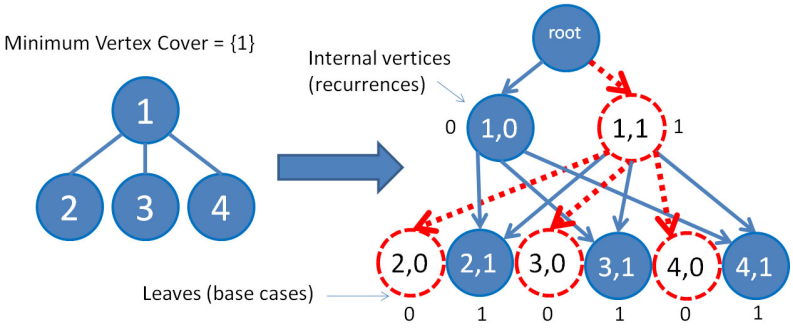


Figure 4.31: The Given General Graph/Tree (left) is Converted to DAG

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 4.31, left, the solution is to take vertex 1 only, because all edges 1-2, 1-3, 1-4 are all incident to vertex 1.

[24]We have mentioned that it is illogical to run DP on a Tree. But the term 'DP on Tree' that actually refers to 'DP on implicit DAG' is already a well-known term.

Now, there are only two possibilities for each vertex. Either it is taken, or it is not. By attaching this 'taken' status to each vertex, we convert the tree into a DAG (see Figure 4.31, 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 will be `min(MVC(root, false), MVC(root, true))`. The solution for the example above is shown as dashed lines in Figure 4.31, right. 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 vertex v is a leaf, false otherwise
    ans = flag;                                                  // 1/0 = taken/not
  else if (flag == 0) {          // if v is not taken, we must take its children
    ans = 0;
    // 'Children' is an Adjacency List that contains the directed version of the
    // tree (parent points to its children; 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;
}
```

Programming Exercises related to DAG:

- 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; implicit DAG; backtracking OK)
  7. **UVa 10350 - Liftless Eme *** (shortest paths; implicit DAG; DP on-the-fly)
     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[25])
  4. **UVa 00988 - Many paths, one destination *** (counting paths in DAG; DP)
  5. **UVa 10401 - Injured Queen Problem *** (counting paths in implicit DAG; DP[26])
  6. UVa 10926 - How Many Dependencies? (counting paths in DAG; DP)
  7. UVa 11067 - Little Red Riding Hood (similar to UVa 825)
  8. **UVa 11957 - Checkers *** (counting paths in DAG; DP)

[25] s: x, y, lastmove, peaksfound; t: try NE/SE
[26] s: col, row; t: next col, avoid 2 or 3 adjacent rows

- Converting General Graph to DAG
  1. UVa 00590 - Always on the Run (s: pos, **day**)
  2. UVa 00907 - Winterim Backpacking Trip (s: pos, **night_left**)
  3. UVa 00910 - TV Game (s: pos, **move_left**)
  4. UVa 10201 - Adventures in Moving ... (s: pos, **fuel**)
  5. **UVa 10243 - Fire; Fire; Fire *** (Min Vertex Cover on Tree; discussed above)
  6. UVa 10543 - Traveling Politician (s: pos, **given**)
  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 on a Grid *** (s: row, col, neg_left, status; t: go down/(left/right))
  10. UVa 11307 - Alternative Arborescence (Min Chromatic Sum, 6 colors are sufficient)
  11. **UVa 11487 - Gathering Food *** (s: r, c, cur food, len; t: 4 dirs)
  12. UVa 11545 - Avoiding Jungle in the Dark (s: cTime, cWTime, cPos, t: move fwd/rest)
  13. UVa 11782 - Optimal Cut (s: id, rem_K; t: take root or try left/right)
  14. LA 3685 - Perfect Service (Kaohsiung06)
  15. LA 4201 - Switch Bulbs (Dhaka08)
  16. SPOJ 101 - Fishmonger (discussed in this section)

4.7.2 Tree

Tree is a special graph with the following characteristics: it has $E = V$-1 (any $O(V+E)$ algorithm 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, post-order traversal. There is no major time speedup as these tree traversal algorithms also runs in $O(V)$, but the codes are simpler. Their pseudo-codes are shown below:

```
pre-order(v)
  visit(v);
  pre-order(left(v));
  pre-order(right(v));
```

```
in-order(v)
  in-order(left(v));
  visit(v);
  in-order(right(v));
```

```
post-order(v)
  post-order(left(v));
  post-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 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 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 4.32.A, the unique path is `5->0->1->3` with weight `3`).

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 weighted graph. However, if the given graph is a tree, the APSP problem becomes *simpler*: Repeat the SSSP on weighted tree V times, one from each vertex. Overall $O(V^2)$.

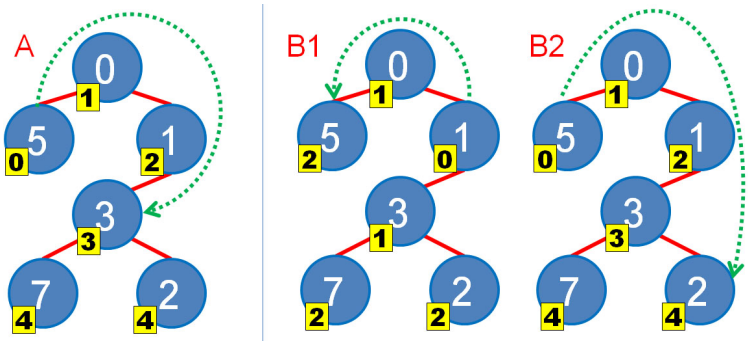


Figure 4.32: A: SSSP/APSP; B1-B2: Diameter

Exercise 4.7.2.1: There is an even faster solution than $O(V^2)$ for the All-Pairs Shortest Paths problem on Weighted Tree. How?

Diameter of Tree

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 of any of these paths is the diameter of the graph. For general graph, we may need $O(V^3)$ Floyd Warshall's algorithm discussed in Section 4.5 plus another $O(V^2)$ all-pairs check to get the maximum distance. However, if the given graph is a 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=5` in Figure 4.32.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=5->0->1->3->y=2` (or `y=7`) with length 4 in Figure 4.32.B2).

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[27].

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!

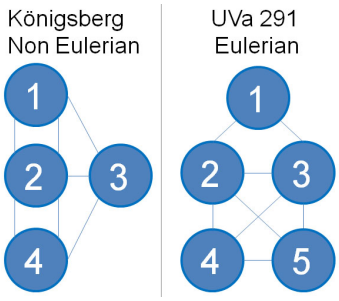


Figure 4.33: Eulerian

Eulerian Graph Check

To check whether an 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 and at most two vertices have odd degrees. This Euler path will start from one of these odd degree vertices and end in the other[28]. 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 4.33.

[27] Compare this property with the *Hamiltonian path/cycle* in TSP (see Section 3.5.2).

[28] Euler path on *directed graph* is also possible: The graph must be connected, has equal in and outdegree vertices, at most one vertex with indegree - outdegree = 1, and at most one vertex with outdegree - indegree = 1.

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 1/0. 1 to say that this edge can still be used, 0 to say that this edge can no longer be used.

```
list<int> cyc; // global variable, 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 (‘used’/‘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 vertex A
  for (list<int>::iterator it = cyc.begin(); it != cyc.end(); it++)
    printf("%d\n", *it);                                        // the Euler tour
```

Example codes: ch4_09_eulerian.cpp; ch4_09_eulerian.java

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 cycle. Tree is also bipartite graph! The most common application is the (bipartite) matching problem, shown below.

Max Cardinality Bipartite Matching (MCBM)

Motivating problem (from TopCoder Open 2009 Qualifying 1 [19, 18]): 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 by pairing $N[0] = 1$ with 4 which results in a prime pair, 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. 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 other item in N. For example, if we pair $N[0] = 1$ with 12, there will be no way to pair the remaining 4 numbers.

Constraints: list N contains an even number of elements (within `[2..50]`, inclusive). Each element of N will be between `1..1000`, inclusive. Each element of N will be distinct.

Although this problem involves prime numbers, it is not a math problem as the elements of N are not more than 1K – there are not too many primes below 1000. 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 + bitmask technique is 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 give edges from `set1` to `set2` if `set1[i] + set2[j]` is prime.

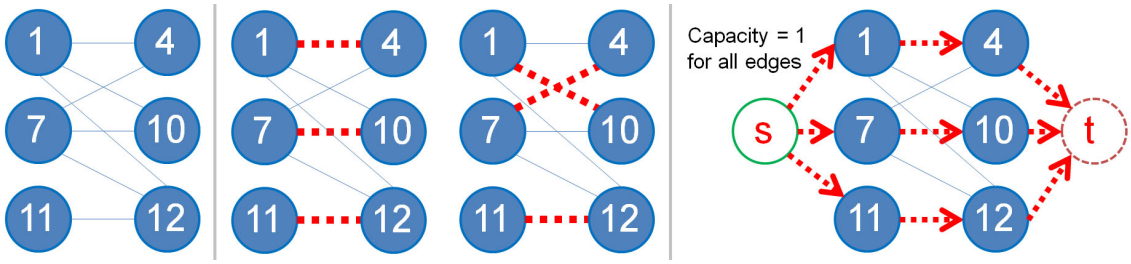


Figure 4.34: Bipartite Matching problem can be reduced to a Max Flow problem

After we build this bipartite graph, the solution is trivial: If the size of `set1` and `set2` are different, 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. 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 4.34, middle).

MCBM can be reduced to the Max Flow problem by assigning a dummy source vertex connected to all vertices in `set1` and a dummy sink vertex connected to all vertices in `set2`. By setting capacities of all edges in this graph to be 1, we force each vertex in `set1` to be matched to only one vertex in `set2`. The Max Flow will be equal to the maximum number of possible matchings on the original graph (see Figure 4.34, right of an example).

Max Independent Set and Min Vertex Cover on Bipartite Graph

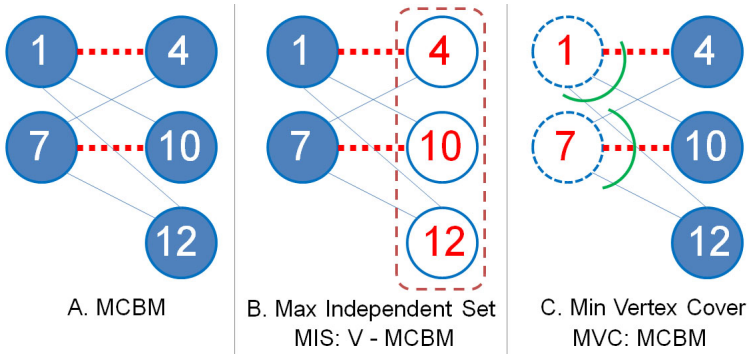


Figure 4.35: MCBM Variants

Independent set is a set S such that every edge of the graph G has at least one endpoint not in S and every vertex not in S has at least one neighbor in S. In Bipartite Graph, the size of the Max Independent Set (MIS) plus the size of the MCBM is equal to the number of vertices V. Or in another word: MIS = V - MCBM. In Figure 4.35.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, {4, 10, 12} are the members of the MIS of this bipartite graph.

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 a MCBM equals the number of vertices in a Min Vertex Cover (MVC). In Figure 4.35.C, we have the same bipartite graph as earlier with MCBM = 2. The MVC is also 2. Indeed, {1, 7} are the members of the MVC of this bipartite graph.

Exercise 4.7.4.1: Solve UVa 10349 - Antenna Placement and UVa 11159 - Factors and Multiples. They are two variant problems involving MCBM.

Min Path Cover on DAG ≈ Max Cardinality Bipartite Matching

Motivating problem: Imagine that the vertices in Figure 4.36.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 min number of taxis that must be deployed to serve *all* passengers?

The answer is two taxis. In Figure 4.36.D, we see one possible 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 solution: $1 \rightarrow 3 \rightarrow 5$ and $2 \rightarrow 4$.

In general, the Min Path Cover (MPC) problem on DAG is described as the problem of finding the min number of paths to cover *each vertex* on DAG $G = (V, E)$.

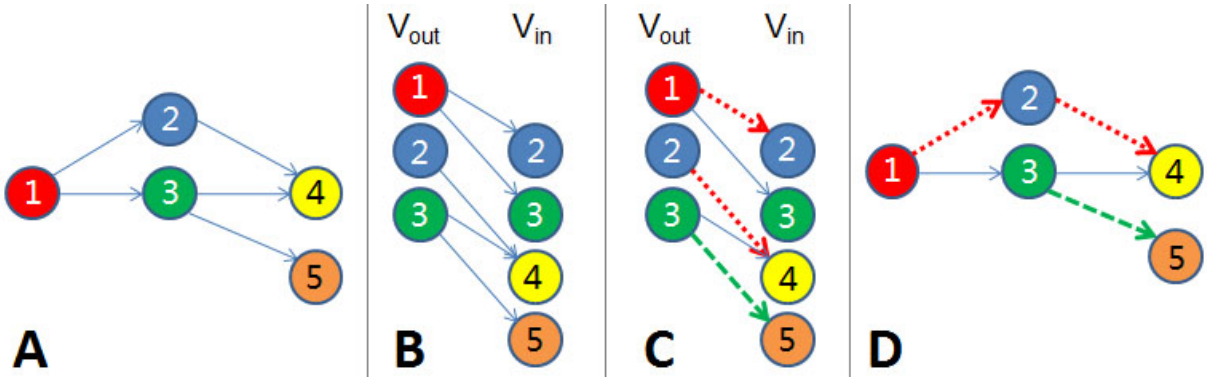


Figure 4.36: Minimum Path Cover on DAG (from LA 3126 [20])

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. Finding a matching on bipartite graph G' forces us to select at most one outgoing edge from $v \in V_{out}$ (similarly for 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) 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 is discussed below. The solution for bipartite matching is polynomial, thus the solution for the MPC in DAG is also polynomial. Note that MPC in general graph is NP-Complete.

Alternating Path Algorithm for Max Cardinality Bipartite Matching

There is a better/faster way to solve the MCBM problem in programming contest rather than going via 'Max Flow route'. We can use the specialized and easy to implement $O(V^2 + VE)$ *alternating path* algorithm. With its implementation handy, all the MCBM problems, including other graph problems that requires MCBM like the Min Path Cover (MPC) in DAG, the Max Independent Set in Bipartite Graph, and the Min Vertex Cover in Bipartite Graph can be solved.

The idea of this alternating path algorithm is to find *augmenting paths*. Let's take a look at a simple graph in Figure 4.37 to understand how this algorithm works.

This algorithm starts with an arbitrary assignment. For our case, we use a greedy assignment. In Figure 4.37.A, we assume that the greedy selection 'wrongly' pair vertex 0 with vertex 3 so that there is 1 matching, but we are unable to find other matching after vertex 0 and 3 are paired.

Then the algorithm tries to find an augmenting path: A path that starts from a free vertex on the left set, alternate between free edge (now on the right set), matched edge (now on the left set again), ..., free edge until the path finally arrives on a free vertex on the right set. In Figure 4.37.B, we can start from a free vertex 2 on the left, go to vertex 3 via a free edge (2-3), go to vertex 0 via a matched edge (3-0), and finally go to vertex 1 via a free edge (1-0). The augmenting path is 2-3-0-1.

Now if we flip the edge status in that augmenting path, i.e. from 'free to matched' and 'matched to free', we will get one more matching overall. See Figure 4.37.C where we flip the status of edges along augmenting path 2-3-0-1. The resulting matching is reflected in Figure 4.37.D.

This algorithm will keep doing this process of finding augmenting paths and flipping them until there is no more augmenting path. The code is shown below.

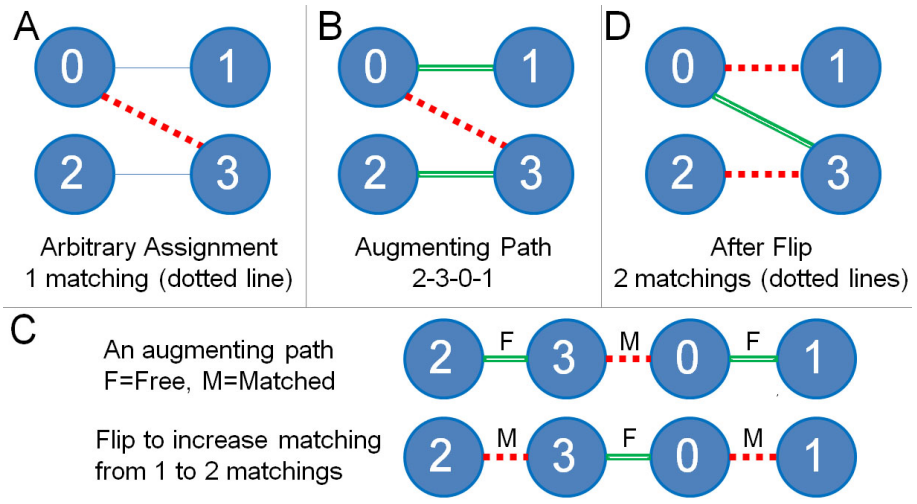


Figure 4.37: Alternating Path Algorithm

```
vi owner, visited;                                          // global variables

int AlternatingPath(int left) {
  if (visited[left]) return 0;                              // to avoid cycling
  visited[left] = true;
  for (int j = 0; j < (int)AdjList[left].size(); j++) {
    ii right = AdjList[left][j];
    // either greedy assignment (if not assigned yet) or recurse
    if (owner[right->first] == -1 || AlternatingPath(owner[right->first])) {
      owner[right->first] = left;
      return 1;                                             // we found one matching
  } }
  return 0;                                                 // no matching
}

// inside int main()
  // build bipartite graph with only directed edge from the left to right set
  // For bipartite graph in Figure 4.37, we have V = 4, num_vertices_on_left = 2
  // AdjList[0] = {(1,1), (3,1)}, note: the second item in the pair is not used
  // AdjList[1] = {}
  // AdjList[2] = {(3,1)}
  // AdjList[3] = {}

  int cardinality = 0;
  owner.assign(V, -1);          // V is the number of vertices in bipartite graph
  for (int left = 0; left < num_vertices_on_left; left++) {
    visited.assign(num_vertices_on_left, 0);      // reset before each recursion
    cardinality += AlternatingPath(left);
  }
  printf("Found %d matchings\n", cardinality);
```

Example codes: ch4_10_mcbm.cpp; ch4_10_mcbm.java

Programming Exercises related to Tree/Eulerian Graph/Bipartite Graph:

- 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 (tree traversal, reconstructing tree from pre + inorder)
  5. UVa 00615 - Is It A Tree? (graph property check)
  6. UVa 00699 - The Falling Leaves (preorder traversal)
  7. UVa 00712 - S-Trees (simple binary tree traversal variant)
  8. UVa 00839 - Not so Mobile (can be viewed as recursive problem on tree)
  9. UVa 10308 - Roads in the North (diameter of tree, discussed in this section)
  10. UVa 10701 - Pre, in and post (reconstructing tree from pre + inorder)
  11. **UVa 10938 - Flea circus *** (Lowest Common Ancestor[29])
  12. **UVa 11615 - Family Tree *** (counting size of subtrees)
  13. **UVa 11695 - Flight Planning *** (cut worst edge along diameter, link 2 centers)
- Eulerian Graph
  1. UVa 00117 - The Postal Worker Rings Once (Euler tour, cost of tour)
  2. UVa 00291 - The House of Santa Claus (Euler tour on a 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 00670 - The Dog Task (MCBM)
  2. UVa 10080 - Gopher II (MCBM)
  3. **UVa 10349 - Antenna Placement *** (Max Independent Set: V - MCBM)
  4. UVa 11045 - My T-Shirt Suits Me (MCBM)
  5. **UVa 11138 - Nuts and Bolts *** (pure MCBM, start from here)
  6. **UVa 11159 - Factors and Multiples *** (Max Independent Set, but ans is the MCBM)
  7. UVa 11418 - Clever Naming Patterns (two layers bipartite graph, use max flow)
  8. UVa 11419 - SAM I AM
  9. LA 2523 - Machine Schedule (Beijing02, Min Vertex Cover)
  10. LA 2696 - Air Raid (Dhaka02, Min Path Cover on DAG $\approx$ MCBM)
  11. LA 3126 - Taxi Cab Scheme (NorthwesternEurope04, MPC on DAG $\approx$ MCBM)
  12. LA 3415 - Guardian of Decency (NorthwesternEurope05, MCBM)
  13. LA 3487 - Duopoly (Hangzhou05, Max *Weighted* Independent Set)
  14. LA 4288 - Cat vs. Dog (NorthwesternEurope08, Max Independent Set)

[29] Let's assume that we have a tree T. The Lowest Common Ancestor (LCA) between two vertices u and v is the lowest vertex in T that has both u and v as its descendants (a vertex can be a descendant of itself). The LCA problem can be reduced to a Range Minimum Query (RMQ) problem (see Section 2.3.3) in linear time.

4.8 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 combine two or more algorithms or to combine an algorithm with some advanced data structures, e.g. using BFS and Dijkstra's together in the same problem to compute shortest path on both weighted and unweighted version of the same graph; 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 8.2.

Of the four special graphs mentioned in this Section 4.7. Trees and DAGs are more popular, especially for IOI contestants. It is *not* rare that Dynamic Programming (DP) on tree or on DAG 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 type of special graph is suitable for the Max Flow problems. However, we reckon that contestants must master the usage of the simpler alternating path algorithm for solving the Max Cardinality Bipartite Matching (MCBM) problem. We have seen in this section that many graph problems are somehow reduceable to MCBM. ICPC contestants should master Bipartite Graph on top of Tree and DAG. IOI contestants do not have to worry with Bipartite Graph as it is still outside IOI 2009 syllabus [10]. 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; etc. When they appear, try to utilize their special properties to speed up your algorithms.

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, **Chu Liu Edmonds algorithm** for Min Cost Arborescence problem, **Tarjan's Offline Lowest Common Ancestor** algorithm, various other matching algorithms (**Hopcroft Karp's** algorithm, **Kuhn Munkres's (Hungarian)** algorithm, **Edmonds's Blossom Shrinking** algorithm), etc.

If you want to increase your winning chance in ACM ICPC, please spend some time to study more graph algorithms/problems beyond[30] 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.

There are ≈ **200 UVa (+ 30 others) programming exercises** discussed in this chapter.
(Only 173 in the first edition, a 33% increase).

There are **49 pages** in this chapter.
(Only 35 in the first edition, a 40% increase).

[30]Interested readers are welcome to explore Felix's paper [12] that discusses maximum flow algorithm for *large* graphs of 411 million vertices and 31 billion edges!

This page is intentionally left blank to keep the number of pages per chapter even.

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 see in this chapter.

Recent ICPCs (especially in Asia) usually contain one or two mathematics problems. Recent IOIs usually do not contain *pure* mathematics tasks, but many tasks require mathematical insights. This chapter aims to prepare contestants in dealing with these mathematics problems.

We are aware that different countries have different emphasis in mathematics training in pre-University education. Thus, for some newbie contestants: 'Binomial Coefficients', 'Catalan Numbers', 'Euler Phi', 'Sieve of Eratosthenes', etc are familiar terms, but for others, these terms do not ring any bell. Perhaps because he has not learnt it before, or perhaps the term is different in his 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.

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 below. These problems are not placed in Section 1.3 as they are Ad Hoc problems with a mathematical flavor. But you can actually jump from Section 1.3 to this section if you prefer to do so. Remember that 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 few lines of code per problem to boost your 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 only 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. Consult Section 3.2 if you need to review various (iterative) Complete Search techniques.

- 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 a set of *square integers* sorted in increasing order: {1, 4, 9, 16, 25, ...} Given an integer X. How many integers in S are less than X? Answer: $\lfloor \sqrt{n} \rfloor$.

- Grid
  These problems involve grid manipulation. The format of the grid can be complex, but the grid will 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.

- Logarithm, Exponentiation, Power
  These problems involve the (clever) usage of `log()` and/or `exp()` function.
  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?

- 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, the operations on polynomial 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 as one of the above.

We suggest that the readers – especially those who are new with mathematics-related problems – to kick start their training programme on solving mathematics-related problems by solving at least 2 or 3 problems *from each sub-category*, especially the ones that we highlight as **must try ***. Note: The problems listed below constitute ≈ 30% of the entire problems in this chapter.

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 Physics (super simple: outputs $2 \times v \times t$)
  3. UVa 10281 - Average Speed (distance = speed × time elapsed)
  4. UVa 10469 - To Carry or not to Carry (super simple if you use `xor`)
  5. **UVa 10773 - Back to Intermediate Math *** (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)

- Mathematical Simulation (Brute Force)
  1. UVa 00100 - The 3n + 1 problem (just follow the description, note that j can be $< i$)
  2. UVa 00371 - Ackermann Functions (similar to UVa 100)
  3. UVa 00382 - Perfection (do trial division)
  4. **UVa 00616 - Coconuts, Revisited *** (brute force up to $\sqrt{n}$, find the pattern)
  5. UVa 00834 - Continued Fractions (do as asked)
  6. UVa 00846 - Steps (uses the sum of arithmetic progression formula[1])
  7. UVa 00906 - Rational Neighbor (start from $d = 1$, compute c, increase until $\frac{a}{b} < \frac{c}{d}$)
  8. UVa 10035 - Primary Arithmetic (simulate; count the number of carry operations)
  9. **UVa 10346 - Peter's Smoke *** (interesting simulation problem)
  10. UVa 10370 - Above Average (compute average, check how many are above it)
  11. UVa 10783 - Odd Sum (input range is very small, just brute force it)
  12. UVa 10879 - Code Refactoring (just use brute force)
  13. **UVa 11130 - Billiard bounces *** (use billiard table reflection technique[2])
  14. UVa 11150 - Cola (similar to UVa 10346, be careful with boundary cases!)
  15. UVa 11247 - Income Tax Hazard (a bit of brute force around the answer to be safe)
  16. UVa 11313 - Gourmet Games (similar to UVa 10346)
  17. UVa 11689 - Soda Surpler (similar to UVa 10346)
  18. UVa 11877 - The Coco-Cola Store (similar to UVa 10346)
  19. UVa 11934 - Magic Formula (just do plain brute-force)
  20. UVa 11968 - In The Airport (average; fabs; if ties, choose the smaller one!)
  21. UVa 11970 - Lucky Numbers (square numbers, divisibility check, bf)
- Finding Pattern or Formula
  1. UVa 00913 - Joana and The Odd Numbers (derive the short formulas)
  2. UVa 10014 - Simple calculations (derive the required formula)
  3. **UVa 10161 - Ant on a Chessboard *** (involves sqrt, ceil...)
  4. UVa 10170 - The Hotel with Infinite Rooms (one liner formula exists)
  5. **UVa 10427 - Naughty Sleepy Boys *** (nums in $[10^{(k-1)}..10^k-1]$ has k digits)
  6. UVa 10499 - The Land of Justice (simple formula exists)
  7. UVa 10509 - R U Kidding Mr. Feynman? (there are only three different cases)
  8. UVa 10666 - The Eurocup is here (analyze the binary representation of X)
  9. UVa 10693 - Traffic Volume (derive the short Physics formula)
  10. UVa 10696 - f91 (very simple formula simplification)
  11. UVa 10940 - Throwing Cards Away II (find pattern with brute force, then use it)
  12. UVa 10970 - Big Chocolate (direct formula exists, or use DP)
  13. UVa 10994 - Simple Addition (formula simplification)
  14. UVa 11202 - The least possible effort (find formula, consider symmetry and flip)
  15. **UVa 11231 - Black and White Painting *** (spot the pattern, get $O(1)$ formula)
  16. UVa 11296 - Counting Solutions to an ... (simple formula exists)
- 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 *** (num of items in row forms AP series; 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)

[1]The sum of arithmetic progression of $\{a_1 = 1, 2, 3, 4, ..., n\}$ is $S_n = n \times (n + 1)/2$. For general case where the delta between two adjacent terms is not one, but d, then $S_n = n/2 \times (2 \times a_1 + (n - 1) \times d)$.

[2]We can simplify problem like this by mirroring the billiard table to the right (and/or top) so that we will only deal with one straight line instead of bouncing lines.

- Number Systems or Sequences
  1. UVa 00136 - Ugly Numbers (use similar technique as UVa 443)
  2. UVa 00138 - Street Numbers (use arithmetic progression formula, precalculated)
  3. UVa 00413 - Up and Down Sequences (simulate the process, array manipulation)
  4. **UVa 00443 - Humble Numbers *** (brute force, 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. **UVa 10042 - Smith Numbers *** (involving prime factorization, sum the digits)
  11. UVa 10101 - Bangla Numbers (follow the problem description carefully)
  12. **UVa 10408 - Farey Sequences *** (generate `(i,j)` pairs s.t. `gcd(i,j) = 1`, sort)
  13. UVa 10930 - A-Sequence (ad-hoc, follow the rules given in problem description)
  14. UVa 11063 - B2 Sequences (check if a number is repeated, be careful with -ve)
  15. UVa 11461 - Square Numbers (answer is $\sqrt{b} - \sqrt{a-1}$)
  16. UVa 11660 - Look-and-Say sequences (simulate, break after j-th character)
  17. LA 4715 - Rating Hazard (Phuket09, Farey Sequence)
- Logarithm, Exponentiation, Power
  1. UVa 00107 - The Cat in the Hat (use logarithm, power)
  2. UVa 00113 - Power Of Cryptography (use exp(ln(x)*y))
  3. **UVa 00701 - Archaelogist's Dilemma *** (use log to count the number of digits)
  4. **UVa 10916 - Factstone Benchmark *** (use logarithm, power)
  5. UVa 11636 - Hello World (uses logarithm)
  6. UVa 11666 - Logarithms (find the formula!)
  7. **UVa 11847 - Cut the Silver Bar *** ($O(1)$ math formula exists: floor of log2(n))
  8. UVa 11986 - Save from Radiation (just $\log_2(N+1)$, use manual check for precision)
- Polynomial
  1. UVa 00392 - Polynomial Showdown (simply follow the orders: output formatting)
  2. **UVa 00498 - Polly the Polynomial *** (straightforward: polynomial evaluation)
  3. **UVa 10268 - 498' *** (polynomial derivation; Horner's rule)
  4. UVa 10302 - Summation of Polynomials (use long double)
  5. **UVa 10586 - Polynomial Remains *** (polynomial division; manipulate coefficients)
  6. UVa 10719 - Quotient Polynomial (polynomial division and remainder)
- Base Number Variants
  1. **UVa 00377 - Cowculations *** (base 4 operations)
  2. **UVa 00575 - Skew Binary *** (base modification)
  3. UVa 10093 - An Easy Problem (try all)
  4. **UVa 10931 - Parity *** (convert decimal to binary, count number of '1's)
  5. UVa 11121 - Base -2 (search for the term 'negabinary')
  6. IOI 2011 - Alphabets (practice task; use the more efficient base 26)
- Just Ad Hoc
  1. UVa 00276 - Egyptian Multiplication (multiplication of Egyptian hieroglyphs)
  2. UVa 00344 - Roman Numerals (convert Roman numerals to decimal and vice versa)
  3. UVa 00759 - The Return of the ... (Roman number + validity check)
  4. **UVa 10137 - The Trip *** (be careful with precision error)
  5. UVa 10190 - Divide, But Not Quite ... (simulate the process)
  6. **UVa 11526 - H(n) *** (brute force up to sqrt(n), find the pattern, avoid TLE)
  7. **UVa 11616 - Roman Numerals *** (Roman numeral conversion problem)
  8. UVa 11715 - Car (physics simulation)
  9. UVa 11816 - HST (simple math, precision required)

5.3 Java BigInteger Class

5.3.1 Basic Features

When the intermediate and/or final result of an integer-based mathematics computation cannot be stored inside the largest built-in integer data type and the given problem does not use any prime-power factorization or modulo arithmetic techniques, 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. This is clearly too large to fit in 64-bit C/C++ `unsigned long long` (or Java `long`).

Exercise 5.3.1.1: Compute the last non zero digit of 25!; Is it possible to use built-in data types?

Exercise 5.3.1.2: Check if 25! is divisible by 9317; Is it possible to use built-in data types?

One way to implement BigInteger library is to store the BigInteger as a (long) string[3]. 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[4]. Fortunately, Java has a BigInteger class that we can use for this purpose. As of 1 August 2011, 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)`, division – `divide(BI)`, remainder – `remainder(BI)`, modulo – `mod(BI)` (slightly different to `remainder(BI)`), division and remainder – `divideAndRemainder(BI)`, power – `pow(int exponent)`, and few other interesting functions discussed later. All are just 'one liner'.

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 simply 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 if you have to write your own BigInteger routines.

[3]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 a number 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 uses a more efficient method than the one shown in this section.

[4]Good news for IOI contestants. IOI tasks usually do not require contestants to deal with BigInteger.

```
import java.util.Scanner;           // Scanner class is inside package java.util
import java.math.BigInteger;     // BigInteger class is 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 ZERO
      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
```

Example code: ch5_01_UVa10925.java

5.3.2 Bonus Features

Java BigInteger class has a few more bonus features that can be useful in programming contests. It happens to have a built-in GCD routine `gcd(BI)`, a modular arithmetic function `modPow(BI exponent, BI m)`, and base number converter: The class's constructor and function `toString(int radix)`. Among these three bonus features, the base number converter is the most useful one. These bonus features are shown with three example problems from UVa online judge. Do not hesitate to use them even if that means that you have to code in Java[5].

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.

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

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 supply > 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));
} } }
```

Example code: ch5_02_UVa10814.java

[5]A note for pure C/C++ programmers: it is good to be a *multi*-lingual programmer.

Modulo Arithmetic

See an example below for LA 4104 - MODEX. We are asked to obtain the value of $x^y \pmod n$.

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

class Main {                                                  /* 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));     // look ma, it's in the library ;)
} } }
```

Example code: ch5_03_LA4104.java

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[6] 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:

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

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;
      String p_str = sc.next();
      BigInteger p = new BigInteger(p_str, b);   // special class's constructor!
      String m_str = sc.next();
      BigInteger m = new BigInteger(m_str, b);      // 2nd parameter is the base
      System.out.println((p.mod(m)).toString(b));      // can output in any base
} } }
```

Example code: ch5_04_UVa10551.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: At the moment (1 August 2011), 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 requires such feature: UVa 10464 and UVa 11821. Try solving these two problems using another library: Java `BigDecimal` class. However, you have to explore this class by yourself.

[6]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 $90 \rightarrow 45(0) \rightarrow 22(1) \rightarrow 11(0) \rightarrow 5(1) \rightarrow 2(1) \rightarrow 1(0) \rightarrow 0(1) = (1011010)_2$ (that is, divide by 2 until 0, then read the remainders from backwards).

Programming Exercises related to BigInteger **NOT**[7] mentioned elsewhere in this chapter.

- Basic Features
  1. UVa 00424 - Integer Inquiry (BigInteger addition)
  2. UVa 00465 - Overflow (read input as BigInteger, add/mult, compare with $2^{31} - 1$)
  3. UVa 00619 - Numerically Speaking (BigInteger)
  4. **UVa 00713 - Adding Reversed Numbers *** (BigInteger + StringBuffer reverse())
  5. UVa 00748 - Exponentiation (BigInteger exponentiation)
  6. UVa 10013 - Super long sums (BigInteger addition)
  7. UVa 10083 - Division (BigInteger + number theory)
  8. UVa 10106 - Product (BigInteger multiplication)
  9. UVa 10198 - Counting (recurrences, BigInteger)
  10. UVa 10494 - If We Were a Child Again (BigInteger division)
  11. UVa 10519 - Really Strange (recurrences, BigInteger)
  12. **UVa 10523 - Very Easy *** (BigInteger addition, multiplication, and power)
  13. UVa 10669 - Three powers (BigInteger is for 3^n, binary rep of set!)
  14. UVa 10925 - Krakovia (BigInteger addition and division)
  15. UVa 11448 - Who said crisis? (BigInteger subtraction)
  16. UVa 11830 - Contract revision (use BigInteger string representation)
  17. **UVa 11879 - Multiple of 17 *** (BigInteger mod, divide, subtract, equals)
  18. LA 3997 - Numerical surprises (Danang07)
  19. LA 4209 - Stopping Doom's Day (Dhaka08, formula simplification, BigInteger)
- Bonus Features
  1. UVa 00290 - Palindroms ←→ smordnilaP (number base conversion; palindrome)
  2. UVa 00343 - What Base Is This? (number base conversion)
  3. UVa 00355 - The Bases Are Loaded (number base conversion)
  4. **UVa 00389 - Basically Speaking *** (number base conv, use Java `Integer` class)
  5. UVa 00446 - Kibbles 'n' Bits 'n' Bits ... (number base conversion)
  6. UVa 00636 - Squares (number base conversion++)
  7. UVa 10464 - Big Big Real Numbers (solvable with Java BigDecimal class)
  8. UVa 10473 - Simple Base Conversion (number base conv, can use C/C++ `strtol`)
  9. UVa 10551 - Basic Remains (BigInteger mod and base conversion)
  10. **UVa 10814 - Simplifying Fractions *** (BigInteger gcd)
  11. UVa 11185 - Ternary (number base conversion: Decimal to base 3)
  12. **UVa 11821 - High-Precision Number *** (solvable with Java BigDecimal class)
  13. LA 4104 - MODEX (Singapore07, BigInteger modPow)

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 or Book of 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.

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.

[7]There will be other programming exercises in other sections in this chapter that also use BigInteger technique.

5.4 Combinatorics

Combinatorics is a branch of *discrete mathematics*[8] concerning the study of **countable** discrete structures. In programming contests, problems involving combinatorics usually titled 'How Many [Object]', 'Count [Object]', etc, although some problem setters 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 patience.

In ICPC[9], if such problem exists in the given problem set, ask one team member 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 (review Section 3.5 for more details). 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 $fib(n) = fib(n-1) + fib(n-2)$ for $n \geq 2$. This generates the following familiar patterns: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, and so on. This pattern sometimes appears in some contest problems which do not mention the term 'Fibonacci' at all, like in some problems shown as exercises below (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. Later in Section 5.9, we will show how to compute Fibonacci numbers in $O(\log n)$ time using efficient matrix exponentiation. As a note, there is an $O(1)$ *approximation* technique to get the *n-th* Fibonacci number. We can compute the closest integer of $\phi^n/\sqrt{5}$ where ϕ (golden ratio) is $((1+\sqrt{5})/2) \approx 1.618$. However this is not so accurate for large Fibonacci numbers.

Exercise 5.4.1.1: Using your calculator, verify if $fib(n) = \phi^n/\sqrt{5}$. Try with small n, check if this 'magic' formula produces $fib(7) = 13$, $fib(9) = 34$, $fib(11) = 89$, etc?

Fibonacci numbers grow very fast and some problems involving Fibonacci have to be solved using Java BigInteger library (see Section 5.3 for a quick solution involving large integers).

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.

Programming Exercises related to Fibonacci Numbers:

1. UVa 00495 - Fibonacci Freeze (very easy with Java BigInteger)
2. UVa 00580 - Critical Mass (related to *Tribonacci* series[10])
3. **UVa 00763 - Fibinary Numbers *** (Zeckendorf representation, greedy, BigInteger)
4. UVa 00900 - Brick Wall Patterns (combinatorics, the pattern is similar to Fibonacci)
5. UVa 00948 - Fibonaccimal Base (Zeckendorf representation, greedy)
6. UVa 10183 - How many Fibs? (get number of Fibs when generating them; BigInteger)
7. **UVa 10334 - Ray Through Glasses *** (combinatorics, use Java BigInteger)

[8] Discrete mathematics is a study of structures that are discrete (like integers {0, 1, 2, ...}, graphs (vertices and edges), logic (true/false)) rather than continuous (like real numbers).

[9] Note that pure combinatorics problem are rare in IOI task although it can be part of a bigger task.

[10] 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 $n \geq 4$.

8. UVa 10450 - World Cup Noise (combinatorics, the pattern is similar to Fibonacci)
9. UVa 10497 - Sweet Child Make Trouble (combinatorics, the pattern is Fibonacci variant)
10. UVa 10579 - Fibonacci Numbers (very easy with Java BigInteger)
11. **UVa 10689 - Yet Another Number ... *** (easy with Pisano period[11])
12. UVa 10862 - Connect the Cable Wires (the pattern ends up very similar to Fibonacci)
13. UVa 11000 - Bee (combinatorics, the pattern is similar to Fibonacci)
14. UVa 11161 - Help My Brother (II) (Fibonacci + median)
15. UVa 11780 - Miles 2 Km (the background of this problem is about Fibonacci numbers)
16. LA 4721 - Nowhere Money (Phuket09, Fibonacci, Zeckendorf Theorem)

5.4.2 Binomial Coefficients

Another classical combinatorics problem is in finding the *coefficients* of the algebraic expansion of powers of a binomial[12]. 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 **{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 $C(n,k)$ 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 ${}^nC_k = {}^nC_{(n-k)}$; during intermediate computations, we divide the numbers first before multiply it with the next number; or use BigInteger technique.

Exercise 5.4.2.1: A frequently used k for $C(n,k)$ is $k = 2$. Show that $C(n,2) = O(n^2)$!

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.

$C(n,0) = C(n,k) = 1$ // 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
```

Programming Exercises related to Binomial Coefficients:

1. UVa 00326 - Extrapolation using a ... (difference table)
2. UVa 00369 - Combinations (be careful with overflow issue)

[11] This problem can be solved easily with 'Pisano (a.k.a Fibonacci) period': "The last/last two/last three/last four digit(s) of a Fibonacci number repeats with a period of 60/300/1500/15000, respectively".

[12] Binomial is a special case of polynomial that only has two terms.

3. **UVa 00485 - Pascal Triangle of Death *** (list the binomial coefficients, BigInteger)
4. UVa 00530 - Binomial Showdown (work with doubles; optimize your computation)
5. UVa 10105 - Polynomial Coefficients ($n!/(n1! \times n2! \times ... \times nk!)$, derivation is complex)
6. **UVa 10219 - Find the Ways *** (compute ${}^nC_k$ with BigInteger, count the length)
7. UVa 10375 - Choose and Divide (the main task is to avoid overflow)
8. **UVa 11955 - Binomial Theorem *** (pure application of this binomial theorem[13], DP)

5.4.3 Catalan Numbers

The number of distinct binary trees with n vertices can be found with the Catalan formula, written using binomial coefficients notation ${}^nC_k$ above as: $Cat(n) = ({}^{(2\times n)}C_n)/(n+1)$; $Cat(0) = 1$.

For example, there are $({}^{(2\times 3)}C_3)/(3+1) = ({}^6C_3)/4 = 20/4 = 5$ distinct binary trees with $n = 3$ vertices. They are shown below.

If we are asked to compute the values of $Cat(n)$ for *several* 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, $Cat(n+1) = \frac{(2n+2)\times(2n+1)}{(n+2)\times(n+1)} \times Cat(n)$.

Programming Exercises related to Catalan Numbers:

1. **UVa 00991 - Safe Salutations *** (Catalan Numbers)
2. **UVa 10007 - Count the Trees *** (answer is $Cat(n) \times n!$, use Java BigInteger)
3. UVa 10223 - How Many Nodes? (Precalculate, only 19 Catalan Numbers $< 2^{32} - 1$)
4. **UVa 10303 - How Many Trees *** (generate $Cat(n)$ as shown above, Java BigInteger)

5.4.4 Other Combinatorics

There are other counting principles and formulas, too many to be discussed in this book.
Try the next few exercises to test/further improve your combinatorics skills.

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, ..., 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

[13] If you use Java BigInteger, use StringBuffer to avoid TLE. Intermediate computations still fit in long long.

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 on your own: Burnside's Lemma, Cayley's Formula, Derangement, Stirling Numbers.

Other Programming Exercises related to Combinatorics:

1. UVa 10079 - Pizza Cutting (derive the one liner formula)
2. UVa 10359 - Tiling (derive formula, use Java BigInteger)
3. UVa 10733 - The Colored Cubes (Burnside's lemma)
4. UVa 10784 - Diagonal (num of diagonals in n-gon $= n*(n-3)/2$, now derive the solution)
5. UVa 10790 - How Many Points of ... (uses arithmetic progression formula)
6. UVa 10843 - Anne's game (Cayley's Formula[14], BigInteger)
7. UVa 10918 - Tri Tiling (there are two related recurrences here)
8. **UVa 11069 - A Graph Problem *** (use Dynamic Programming)
9. UVa 11115 - Uncle Jack (N^D, use Java BigInteger)
10. UVa 11204 - Musical Instruments (only 1st choice matters)
11. **UVa 11310 - Delivery Debacle *** (requires DP[15])
12. **UVa 11401 - Triangle Counting *** (spot the pattern, coding is easy)
13. UVa 11480 - Jimmy's Balls (try all r, but simpler formula exists)
14. UVa 11554 - Hapless Hedonism (similar to UVa 11401)
15. UVa 11597 - Spanning Subtree (uses knowledge of graph theory, the answer is very trivial)
16. UVa 11609 - Teams ($N \times 2^{N-1}$, use Java BigInteger for the modPow part)
17. LA 3904 - Tile Code (Seoul07, Combinatorics)
18. LA 4847 - Binary Search Tree (Daejeon10, Combinatorics, Recurrence, Graph Theory)

Profile of Algorithm Inventors

Eratosthenes of Cyrene (≈ 300-200 years BC) was a Greek mathematician. He invented geography, did measurements of the circumference of earth, and invented a simple algorithm to find 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 (≈ 200-300 AD) was an Alexandrian Greek mathematician. He did a lot of study in algebra. One of his works in this book is the Linear Diophantine Equations.

[14]This formula counts the number of spanning trees of a graph with n vertices, which is n^{n-2}.

[15]Let `dp[i]` be the number of ways the cakes can be packed for a box `2×i`. Note that it is possible to use the 2 L shaped cakes to form a 2x3 shape.

5.5 Number Theory

Mastering as many topics as possible in the field of *number theory* is important as some mathematics problems becomes 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: $\{2, 3, 4, 5, \ldots\}$ 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 [31]). 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[16] $104729, 1299709, 15485863, 179424673, 2147483647$, etc.

Prime number is an important topic in number theory and the source for many programming problems[17]. 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 of course 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 improvement is to test if N is divisible by a *divisor* $\in$ `[2..`$\sqrt{N}$`]`, i.e. we stop when the *divisor* is already greater than $\sqrt{N}$. Reason: If N is divisible by p, then $N = p \times q$. If q were smaller than p, then q or a prime factor of q would have divided N earlier. This is $O(\sqrt{N})$ which is already much faster than the previous version, but can still be improved to be twice faster.

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[18] which is already good enough 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 [31] 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 algorithm is called 'Sieve of *Eratosthenes*' invented by Eratosthenes of Alexandria. It works as follows.

First, it 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[19] 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

[16] 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.

[17] 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**).

[18] 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.

[19] Common sub-optimal implementation is to start from 2 × i instead of i × i, but the difference is not that much.

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 + ... + 1/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 small primes $\leq 10K$ using the sieve is fast (our library 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, more efficient 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);  // also add this vector containing 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
```

Example code (first part): ch5_05_primes.cpp; ch5_05_primes.java

---

Programming Exercises related to Prime Numbers:

1. UVa 00406 - Prime Cuts (sieve; take the middle ones)
2. **UVa 00543 - Goldbach's Conjecture *** (sieve; complete search[20])
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 10200 - Prime Time (complete search, test[21] $\forall n \in$ [a..b] if isPrime($n^2 + n + 41$))
9. UVa 10235 - Simply Emirp (case analysis; not prime/prime/emirp:prime+reversed prime)

[20] Christian Goldbach's conjecture (updated by Leonhard Euler) says that every even number greater than or equal to 4 can be expressed as the sum of two prime numbers.

[21] 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.

10. UVa 10311 - Goldbach and Euler (case analysis, complete search, see UVa 543)
11. UVa 10394 - Twin Primes (sieve; get adjacent primes)
12. UVa 10490 - Mr. Azad and his Son (a bit Ad Hoc; the answers can be precalculated)
13. **UVa 10539 - Almost Prime Numbers *** (sieve; get almost primes[22]; sort; bsearch)
14. UVa 10650 - Determinate Prime (find 3 consecutive primes that are uni-distance)
15. **UVa 10738 - Riemann vs. Mertens *** (modified sieve[23]; complete search)
16. UVa 10852 - Less Prime (sieve; $p = 1$, find the first prime number $\geq \frac{n}{2} + 1$)
17. UVa 10924 - Prime Words (check if sum of letter values is a prime)
18. UVa 10948 - The Primary Problem (another Goldbach's conjecture problem, see UVa 543)
19. UVa 11287 - Pseudoprime Numbers (yes if !isPrime(p)+a.modPow(p, p) = a; BigInteger)
20. UVa 11752 - The Super Powers (base: 2 to $\sqrt[4]{2^{64}}$, composite power, sort, output)
21. LA 3399 - Sum of Consecutive ... (Tokyo05, Prime Numbers)

5.5.2 Greatest Common Divisor (GCD) & Least Common Multiple (LCM)

The Greatest Common Divisor (GCD) of two integers (a, b) denoted by $gcd(a, b)$, is defined as the largest positive integer d such that $d \mid a$ and $d \mid b$ where $x \mid y$ implies that x divides y. Example of GCD: $gcd(4, 8) = 4$, $gcd(10, 5) = 5$, $gcd(20, 12) = 4$. One practical usage of GCD is to simplify fractions, e.g. $\frac{4}{8} = \frac{4/gcd(4,8)}{8/gcd(4,8)} = \frac{4/4}{8/4} = \frac{1}{2}$.

To find the GCD between two integers is an easy task with an effective *Euclid* algorithm [31, 3] which can be implemented as a one liner code (see below). Thus finding the GCD is usually not the actual issue in a Math-related contest problem, but just part of the 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: $lcm(4, 8) = 8$, $lcm(10, 5) = 10$, $lcm(20, 12) = 60$. It has been shown [31] 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))); }  // divide bef multiply!
```

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 $n = max(a, b)$.

Programming Exercises related to GCD and/or LCM:

1. UVa 00106 - Fermat vs. Phytagoras (opt brute force; GCD to get relatively prime triples)
2. UVa 00332 - Rational Numbers from ... (use GCD to simplify fraction)
3. UVa 00408 - Uniform Generator (good choice if step < mod & GCD(step, mod) == 1)
4. UVa 00412 - Pi (complete search; use GCD to find elements with no common factor)
5. UVa 10193 - All You Need Is Love (convert input to decimal, decide if GCD > 1 or not)
6. **UVa 10407 - Simple Division *** (subtract the set s with s[0], find gcd)
7. **UVa 10892 - LCM Cardinality *** (no of div pairs of N: (m, n) s.t. gcd(m, n) = 1)

[22] In this problem, 'almost prime' is defined as non prime numbers that is 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'.

[23] In this problem, one of the requirement is to count the number of prime factors $\forall i \in$ `[2..1M]`. Instead of doing this frontally - which is slow, we can solve this 'backwards' by modifying the Sieve algorithm. Initially, the number of prime factors for each integer is 0, that is `numPF[i] = 0`. Then, starting from $i = 2$, if `numPF[i] = 0` (a prime), instead of crossing out its multiples, we *increase* `numPF[multiple of i]` by 1 (that is, we add one more prime factor to this multiple of i). We repeat until $i = 1M$.

8. UVa 11388 - GCD LCM (must understand the relationship between GCD and LCM)
9. UVa 11417 - GCD (brute force, input is small)
10. **UVa 11827 - Maximum GCD *** (simple; find GCD of many numbers, small input)

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. $fac(n) = 2 \times 3 \times 4 \times \ldots \times (n-1) \times n$ (loop from 2 to n, usually 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 (refer back to Section 5.3), work with the prime factors of a factorial (see the next two subsections, Section 5.5.5), or get the results modulo a smaller number (see the next four subsections, Section 5.5.7).

Programming Exercises related to Factorial:

1. **UVa 00324 - Factorial Frequencies *** (count digits of n!, up to 366!, Java BigInteger)
2. UVa 00568 - Just the Facts (can use Java BigInteger, slow but AC)
3. **UVa 00623 - 500 (factorial) *** (easy with Java BigInteger, appear as teaser in Ch 1)
4. UVa 10220 - I Love Big Numbers (use Java BigInteger; precalculate)
5. UVa 10323 - Factorial. You Must ... (overflow: n>13/-odd n; underflow[24]: n<8/-even n)
6. **UVa 10338 - Mischievous Children *** (use long long to store up to 20!)

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, $N = 240 = 2 \times 2 \times 2 \times 2 \times 3 \times 5 = 2^4 \times 3 \times 5$ (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 factor PF. We can keep doing this until eventually $N' = 1$. To speed up the process even further, we utilize the divisibility property[25] 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 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 $2^{10} \times 3^4 \times 5 \times 7^4 \times 11 \times 13$.

Exercise 5.5.4.1: Examine the given code below. 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).

[24] Actually, this problem is not mathematically correct as factorial of negative number is not defined.
[25] If d is a divisor of n, then so is $\frac{n}{d}$, but d and $\frac{n}{d}$ cannot *both* be greater than $\sqrt{n}$.

```
vi primeFactors(ll N) {          // remember: vi is vector<int>, ll is long long
  vi factors;
  ll PF_idx = 0, PF = primes[PF_idx]; // using PF = 2, then 3,5,7,... is also ok
  while (N != 1 && (PF * PF <= N)) {   // stop at sqrt(N), but N can get smaller
    while (N % PF == 0) { N /= PF; factors.push_back(PF); }    // remove this PF
    PF = primes[++PF_idx];                                  // only consider primes!
  }
  if (N != 1) factors.push_back(N);     // special case if N is actually a prime
  return factors;   // if N does not fit in 32-bit integer and is a prime number
}                        // then 'factors' will have to be changed to vector<ll>

// inside int main(), assuming sieve(1000000) has been called before
  vi res = primeFactors(2147483647);           // slowest, 2147483647 is a prime
  for (vi::iterator i = res.begin(); i != res.end(); i++) printf("> %d\n", *i);

  res = primeFactors(136117223861LL);   // slow, 2 large pfactors 104729*1299709
  for (vi::iterator i = res.begin(); i != res.end(); i++) printf("# %d\n", *i);

  res = primeFactors(142391208960LL);            // faster, 2^10*3^4*5*7^4*11*13
  for (vi::iterator i = res.begin(); i != res.end(); i++) printf("! %d\n", *i);
```

Example code (second part): ch5_05_primes.cpp; ch5_05_primes.java

---

Programming Exercises related to Finding Prime Factors with Optimized Trial Divisions:

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 number)
4. **UVa 11466 - Largest Prime Divisor *** (use efficient sieve; get largest prime factors)

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 $n!$? $(0 \leq n, m \leq 2^{31}-1)$". 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: $6! = 2\times3\times4\times5\times6 = 2\times3\times(2^2)\times5\times(2\times3) = 2^4\times3^2\times5$. 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: *Without* computing the actual integers, determine what is the GCD and LCM of $(2^6 \times 3^3 \times 97^1, 2^5 \times 5^2 \times 11^2)$?

Programming Exercises related to 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); precalculate)
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 (involving prime-power factorization; combinatorics)
12. **UVa 11889 - Benefit *** (LCM, involving prime factorization)
13. LA 2195 - Counting Zeroes (Dhaka06, prime numbers)

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 are welcome to explore Chapter 7: "Multiplicative Functions" of [31].

1. `numPF(N)`: Count the number of *prime factors* of `N`

   A straightforward 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 (N != 1 && (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`

   Exercise 5.5.6.1: `numDiffPF(N)` and `sumPF(N)` are similar to `numPF(N)`. Implement them!

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. If you are curious, the 12 divisors are: $\{1, \underline{\mathbf{2}}, \underline{\mathbf{3}}, 4, \underline{\mathbf{5}}, 6, 10, 12, 15, 20, 30, 60\}$. The prime factors of 12 are **highlighted**. See that N has more divisors than prime factors.

5. `sumDiv(N)`: *Sum* the divisors of `N`

   In the previous example, the 12 divisors of $N = 60$ are $\{1, 2, 3, 4, 5, 6, 10, 12, 15, 20, 30, 60\}$. 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 $\frac{a^{i+1}-1}{a-1} \times \frac{b^{j+1}-1}{b-1} \times ... \times \frac{c^{k+1}-1}{c-1}$.
   Let's check. $N = 60 = 2^2 \times 3^1 \times 5^1$, `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 numDiv(ll N) {
  ll PF_idx = 0, PF = primes[PF_idx], ans = 1;          // start from ans = 1
  while (N != 1 && (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;
}

ll sumDiv(ll N) {
  ll PF_idx = 0, PF = primes[PF_idx], ans = 1;          // start from ans = 1
  while (N != 1 && (PF * PF <= N)) {
    ll power = 0;
    while (N % PF == 0) { N /= PF; power++; }
    ans *= ((ll)pow((double)PF, power + 1.0) - 1) / (PF - 1);      // formula
    PF = primes[++PF_idx];
  }
  if (N != 1) ans *= ((ll)pow((double)N, 2.0) - 1) / (N - 1);     // last one
  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-1]`, and increases the `counter` if $gcd(i, N) = 1$. This algorithm is slow for a large N.

   A better algorithm is the Euler's Phi (Totient) function. The Euler's Phi $\varphi(N)$ is a function to compute the solution for the problem posed above. $\varphi(N) = N \times \prod_{PF} (1 - \frac{1}{PF})$, 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 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 (N != 1 && (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;
}
```

Example code (third part): `ch5_05_primes.cpp`; `ch5_05_primes.java`

Programming Exercises related to Functions involving Prime Factors:

1. **UVa 00294 - Divisors *** (`numDiv(N)`)
2. UVa 00884 - Factorial Factors (`numPF(N)`; precalculate)
3. **UVa 10179 - Irreducible Basic Fractions *** (`EulerPhi(N)`)
4. UVa 10299 - Relatives (`EulerPhi(N)`)
5. UVa 10699 - Count the Factors (`numDiffPF(N)`)
6. UVa 10820 - Send A Table (`a[i] = a[i - 1] + 2 * EulerPhi(i)`)
7. UVa 11064 - Number Theory (`N - EulerPhi(N) - numDiv(N)`)
8. UVa 11086 - Composite Prime (find numbers N with `numPF(N) == 2`)
9. UVa 11226 - Reaching the fix-point (`sumPF(N)`; get length; DP)
10. UVa 11327 - Enumerating Rational ... (pre-calculate `EulerPhi(N)`, $\forall N =$ `[1..200K]`)
11. **UVa 11728 - Alternate Task *** (`sumDiv(N)`)
12. LA 4340 - Find Terrorists (Amrita08, `numDiv(N)`)

5.5.7 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 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 reach of 64-bit integers. 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, and then quickly perform modulo operation to the intermediate result by 131071. If the final result is 0, then the *actual number in binary* (which we never compute in its entirety), is divisible by 131071.

Exercise 5.5.7.1: Which statements are valid? Note: '%' is a symbol of modulo operation.
1). `(a + b - c) % s = ((a % s) + (b % s) - (c % 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.

Programming Exercises related to Modulo Arithmetic:

1. UVa 00128 - Software CRC ((a*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)
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 shallow *** (discussed in this section)
6. **UVa 10212 - The Last Non-zero Digit *** (there is a non number theory solution[26])
7. UVa 10489 - Boxes of Chocolates (keep intermediate computation small with modulo)

[26]Multiply numbers from N down to N-M+1, repeatedly use % 10 to discard the trailing zero(es), then use % 1 Billion to only memorize the last few (maximum 9) non zero digits.

5.5.8 Extended Euclid: Solving Linear Diophantine Equation

Motivating problem: Suppose a housewife buys apples and oranges at a total cost of 8.39 SGD. If an apple is 25 cents and an orange is 18 cents, how many of each type of fruits 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 for the Linear Diophantine Equation 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; }
  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 $25 \times (-5) + 18 \times 7 = 1$.

Multiplying the left and right hand side of the equation above by $839/gcd(25, 18) = 839$, we have:
$25 \times (-4195) + 18 \times 5873 = 839$.
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.

Programming Exercises related to Extended Euclid:

1. **UVa 10090 - Marbles *** (use solution for Linear Diophantine Equation)
2. **UVa 10104 - Euclid Problem *** (pure problem involving Extended Euclid)
3. UVa 10633 - Rare Easy Problem (can be modeled as Linear Diophantine Equation[27])
4. **UVa 10673 - Play with Floor and Ceil *** (uses Extended Euclid)

---

[27] 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.

5.5.9 Other Number Theoretic Problems

There are many other number theoretic problems that cannot be discussed one by one in this book. To close this section about number theory, we list down few more programming problems related to number theory that are not classified under one of the categories above.

Programming Exercises related to 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 Newbies *** (the solving strategy is already given)

5.6 Probability Theory

Probability Theory is a branch of mathematics dealing with the analysis of random phenomena. Although an event like an individual coin toss is random, if it is repeated many times the sequence of random events will exhibit certain statistical patterns, which can be studied and predicted. In programming contests, problems involving probability are either solvable with:

- Closed-form formula. For these problems, one has to derive the required formula. Examples: UVa 10056 - What is the Probability?, UVa 10491 - Cows and Cars, etc.
- Exploration of the search space to count number of events (usually harder to count; may deal with combinatorics, Complete Search, or Dynamic Programming) over the countable sample spaces (usually much simpler to count). Examples:
  - 'UVa 12024 - Hats' can be solved via brute-force by trying all $n!$ permutations and see how many times the required events appear over $n!$ (n factorial). However, a more math-savvy contestant can use this formula instead: $A_n = (n-1) \times (A_{n-1} + A_{n-2})$.
  - 'UVa 10759 - Dice Throwing' can be solved using DP (lots of overlapping subproblems) to count the number of events (n dices summed to $\geq x$) over 6^n.

Programming Exercises about Probability Theory:

1. UVa 00474 - Heads Tails Probability (actually this is just a `log` & `pow` exercise)
2. UVa 00545 - Heads (use logarithm, power, similar to UVa 474)
3. UVa 10056 - What is the Probability? (get the closed form formula)
4. UVa 10238 - Throw the Dice (similar to UVa 10759, F values, Java BigInteger)
5. UVa 10328 - Coin Toss (DP, 1-D state, Java BigInteger)
6. **UVa 10491 - Cows and Cars *** (There are two ways to get a car[28])
7. **UVa 10759 - Dice Throwing *** (DP, state: (dice_left, score), try 6 values, gcd)
8. UVa 11181 - Probability (bar) Given (iterative brute force, try all possibilities)
9. UVa 11500 - Vampires (Gambler's Ruin Problem)
10. UVa 11628 - Another lottery (p[i] = ticket[i] / total; gcd to simplify fraction)
11. **UVa 12024 - Hats *** ($n \leq 12$; run brute force in 15 mins; then precalculate)

[28] Either pick a cow first, then switch to a car; or pick a car first, and then switch to another car.

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 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 $f(x) = (7x+5)\%12$ 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 $\mu = 0$ (always the case for this problem) and $\lambda = 6$ as $x_0 = x_{\mu+\lambda} = x_{0+6} = x_6 = 4$. The iterated function values cycles from index 6 onwards.

5.7.1 Solution 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 $O(\mu + \lambda)$ but also requires at least $O(\mu + \lambda)$ space to store past values.

For many cycle-finding problems with $\mu = 0$ and rather large S (and likely large λ), we can use $O(\lambda)$ space C++ STL `set`/Java `TreeSet` to store/check past values in $O(\log \lambda)$ time.

For other cycle-finding problems with $\mu = 0$ and relatively small S (and likely small λ), we may use the $O(|S|)$ space Direct Addressing Table to store/check past values in $O(1)$ time.

5.7.2 Floyd's Cycle-Finding Algorithm

There is a better algorithm called Floyd's cycle-finding algorithm that runs in the same $O(\mu + \lambda)$ time complexity but *only* uses $O(1)$ memory space – much smaller than the simple version above. This algorithm is also called 'the tortoise and hare (rabbit)' algorithm. It has three components.

Efficient Way to Detect a Cycle: Finding $k\lambda$

step	x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}	x_{11}	x_{12}
	4	9	8	1	0	5	4	9	8	1	0	5	4
Init	TH												
1		T	H										
2			T		H								
3				T			H						
4					T				H				
5						T					H		
6							<u>T</u>						<u>H</u>

Table 5.1: Part 1: Finding $k\lambda$, $f(x) = (7x + 5)\%12$, $x_0 = 4$

Note that for any $i \geq \mu$, $x_i = x_{i+k\lambda}$, where $k > 0$, e.g. in Table 5.1, $x_0 = x_{0+1\times6} = x_6 = x_{0+2\times6} = x_{12} = 4$. If we pick $i = k\lambda$, we get $x_i = x_{i+i} = x_{2i}$. Floyd's cycle finding algorithm exploits this.

The Floyd's cycle-finding algorithm maintains two pointers called 'tortoise' at x_i and 'hare' 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[29] 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 $k\lambda$ (multiple of λ). We will determine the actual λ from $k\lambda$ using the next two steps. In Table 5.1, when $i = 6$, we have $x_6 = x_{12} = x_{6+\underline{\mathbf{6}}} = x_{6+k\lambda} = 4$. So, $k\lambda = 6$. In this example, we will see below that k is eventually 1, so $\lambda = 6$ 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. When tortoise and hare points to the same value, we have just find the *first* repetition of length $k\lambda$. Since $k\lambda$ is a multiple of λ, it must be true that $x_\mu = x_{k\lambda+\mu}$. The first time we encounter the first repetition of length $k\lambda$ is also the value of the μ. In Table 5.2, it happens that $\mu = 0$. In fact, many case of cycle-finding problems have $\mu = 0$.

step	x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}	x_{11}	x_{12}
	4	9	8	1	0	5	4	9	8	1	0	5	4
1	H						T						

Table 5.2: Part 2: Finding μ

Finding λ

Once we got μ, we can set tortoise to point to x_μ and hare to point to $f(x_\mu)$. Now, we set tortoise to stay at x_μ and move hare iteratively to the right one by one. Hare will point to a value that is the same as x_μ for the *first* time after λ steps. In Table 5.3, after hare moves 6 times, $x_\mu = x_0 = x_{\mu+\lambda} = x_{0+6} = x_6 = 4$. So, $\lambda = 6$.

step	x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}	x_{11}	x_{12}
	4	9	8	1	0	5	4	9	8	1	0	5	4
1	T	H											
2	T		H										
3	T			H									
4	T				H								
5	T					H							
6	T						H						

Table 5.3: Part 3: Finding λ

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 element/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; tortoise = mu; hare = f(tortoise);
  while (tortoise != hare) { hare = f(hare); lambda++; }
  return ii(mu, lambda);
}
```

[29]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 $x_i = f(f(x_i))$.

Example code: ch5_06_UVa350.cpp; ch5_06_UVa350.java

Programming Exercises related to Cycle-Finding:

1. UVa 00202 - Repeating Decimals (do fraction expansion digit by digit until it cycles)
2. UVa 00275 - Expanding Fractions (very similar to UVa 202, only different in output format)
3. **UVa 00350 - Pseudo-Random Numbers *** (direct application of Floyd's cycle-finding)
4. UVa 00944 - Happy Numbers (similar to UVa 10591)
5. UVa 10162 - Last Digit (cycle after 100 steps, Java Big Integer to read input, precalc)
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 sequence (cycle-finding, eval Reverse Polish `f` with `stack`)
9. **UVa 11053 - Flavius Josephus Reloaded *** (cycle-finding, the answer is $N - \lambda$)
10. UVa 11549 - Calculator Conundrum (repeat squaring with limited digits until it cycles[30])
11. **UVa 11634 - Generate random numbers *** (use DAT of size 10K, extract digits[31])

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 Games** – a mathematical way of saying that if one player wins, then the other player losses. 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.

5.8.1 Decision Tree

The common question asked in programming problems related to game theory is whether the starting player has a winning move assuming both players are doing **Perfect Play**. That is, each player always choose the most optimal choice available to him.

One solution is to write a recursive code to explore the **Decision Tree** of the game (also called the Game Tree). If there is no overlapping subproblem, pure recursive backtracking is suitable. Otherwise, Dynamic Programming is needed. Each node describes the current player and the current state of the game. Each node is connected to all other nodes legally reachable from that node according to the game rules. The root node describes the starting player and the initial game state. If the game state at a leaf node is a winning state, it is a win for the current player (and a lose for the other player). At an internal node of the starting player, choose a node that guarantees his win with the largest margin (or if that is not possible, he choose a node where he loses with the smallest margin). This is called the **Minimax** strategy.

For example, in UVa 10368 - Euclid's Game, there are two players: Stan and Ollie. The state of the game is two positive integers (a, b). Each player can subtracts any positive multiple of the lesser of the two numbers from the greater of the two numbers, provided that the resulting number must be nonnegative. 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. See the decision tree below for initial game state $a = 34$ and $b = 12$. S is 'Stan', O is 'Ollie'.

[30] 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.

[31] The programming trick to square 4 digits 'a' and get the resulting middle 4 digits is `a = (a * a / 100) % 10000`.

```
// Stan has 2 choices: subtract 34 with 12 or 24
// His optimal move is to do 34-24, as it is a sure lose for Ollie (win for Stan)

                          (S, a=34, b=12)
     34-1*12 = 34-12 = 22 /             \ 34-2*12 = 34-24 = 10
(O, a=22, b=12) Win  for O               (O, a=12, b=10) Lose for O
 | 22-1*12 = 10      ^                    | 12-1*10 = 2       ^
(S, a=12, b=10) Lose for S               (S, a=10, b= 2) Win  for S
 | 12-1*10 = 2       ^                     10-5*2 = 0
(O, a=10, b= 2) Win  for O
   10-5*2 = 0
```

5.8.2 Mathematical Insights to Speed-up the Solution

However, not all game theory problems can be solved directly. Sometimes, if the problems involve numbers, we have to come up with some number theoretic insights to speed up the computation.

For example, in UVa 847 - A multiplication game, there are two players: Stan and Ollie again. The state of the game is an integer p. Each 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 and the initial game state is $p = 1$.

It turns out that the optimal strategy for Stan to win is to *always* multiply p with 9 (the largest possible) so that he can reach $p \geq n$ as fast as possible (the target number) while Ollie will *always* multiply p with 2 to prevent Stan from reaching $p \geq n$. Such optimization insights can be obtained by observing the pattern found in the decision tree. Without this insight, the decision tree will have a *huge* branching factor of 8 (as there are 8 possible numbers to choose from between 2 to 9). Recursive backtracking solution will get a TLE verdict.

5.8.3 Nim Game

There is a special game that is worth mentioning: the **Nim** game[32]. 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 $n_1 \wedge n_2 \wedge ... \wedge n_k$ 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 Move *** (Tic-Tac-Toe, minimax, backtracking)
3. UVa 10165 - Stone Game (Nim game, pure application of Sprague-Grundy theorem)
4. UVa 10368 - Euclid's Game (minimax, backtracking, discussed above)
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[33])
8. **UVa 11489 - Integer Game *** (game theory, reducible to simple math)

---

[32] While the general form of two player games discussed earlier is inside the IOI syllabus, this special game is currently excluded [10].

[33] 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.

5.9 Powers of a (Square) Matrix

In this section, we discuss a special case of matrix[34]: the *square matrix*[35]. To be precise, we discuss a special operation of square matrix: the *powers of a square matrix*. Mathematically, we say that $M^0 = I$ and $M^p = \prod_{i=1}^{p} M$. I is the *Identity* matrix[36] and p is the asked 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* Fibonacci number $fib(p)$ in $O(\log p)$ time instead of $O(p)$.
  Imagine if $p = 2^{30}$. $O(p)$ solution will get TLE whereas $\log_2(2^{30})$ is just 30 steps.
  Note: if we need *all* $fib(n)$ for all $n \in$ `[1..n]`, use $O(n)$ DP solution instead.
  This is achievable by using the following equality:

```
[ 1 1 ]^p   [ fib(p+1) fib(p)   ]
[ 1 0 ]   = [ fib(p)   fib(p-1) ]
```

  For example, to compute $fib(11)$, we simply multiply the Fibonacci matrix 11 times, i.e. raise it to the power of 11.

```
[ 1 1 ]^11   [ 144 89 ]   [ fib(12) fib(11) ]
[ 1 0 ]    = [  89 55 ] = [ fib(11) fib(10) ]
```

- Compute number of paths of length L of a graph stored in an Adjacency Matrix in $O(n^3 \log L)$.

  Adjacency Matrix is a square matrix. 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.

```
            0->1 with length 1 -> possible: 0->1 (only 1 path)
0--1        0->1 with length 2 -> impossible
   |        0->1 with length 3 -> possible: 0->1->2->1 (0->1->0->1)
   2--3     0->1 with length 4 -> impossible
            0->1 with length 5 -> possible: 0->1->2->3->2->1 (and 4 others)

M = [ 0 1 0 0 ]   M^2 = [ 1 0 1 0 ]   M^3 = [ 0 2 0 1 ]   M^5 = [ 0 5 0 3 ]
    [ 1 0 1 0 ]         [ 0 2 0 1 ]         [ 2 0 3 0 ]         [ 5 0 8 0 ]
    [ 0 1 0 1 ]         [ 1 0 2 0 ]         [ 0 3 0 2 ]         [ 0 8 0 5 ]
    [ 0 0 1 0 ]         [ 0 1 0 1 ]         [ 1 0 2 0 ]         [ 3 0 5 0 ]
```

5.9.1 The Idea of Efficient Exponentiation

For the sake of argument, let's assume that built-in function like *pow(A, p)* or other related functions that can raise a number A to a certain *integer* power p does not exist. Now, if we do exponentiation 'by definition', we will have an inefficient $O(p)$ solution, especially if p is large[37].

```
int normalExp(int base, int p) {    // for simplicity, we just use int data type
  int ans = 1;
  for (int i = 0; i < p; i++) ans *= base;                          // O(p)
  return ans; }
```

[34] 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.

[35] A square matrix is a matrix with the same number of rows and columns, i.e. $n \times n$.

[36] Identity matrix is a matrix with all cells filled with zeroes except that cells along the main diagonal are all ones.

[37] If you encounter input size of 'gigantic' value in programming problems, like 1B, the problem setter is *usually* looking for a logarithmic solution. Notice that $\log_2(1B) \approx \log_2(2^{30})$ is still just 30!

There is a better way that uses Divide & Conquer principle. We can express A^p as:

$A^0 = 1$ and $A^1 = A$ (two base cases).
$A^p = A^{p-1} \times A$ if p is odd.
$A^p = (A^{p/2})^2$ if p is even. As this approach keep 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 is shown below:

```
int fastExp(int base, int p) {                                          // O(log p)
       if (p == 0) return 1;
  else if (p == 1) return base;
  else {           int res = fastExp(base, p / 2); res *= res;
                   if (p % 2 == 1) res *= base;
                   return res; } }
```

5.9.2 Square Matrix Exponentiation

We can use the same efficient exponentiation technique shown above to perform square matrix exponentiation in $O(n^3 \log p)$. Each matrix multiplication is $O(n^3)$. In case of 2×2 Fibonacci matrix, this is just 8 multiplications, thus we can compute $fib(p)$ in $O(\log p)$. The *iterative* implementation (for comparison with the recursive implementation shown earlier) is shown below:

```
#define MAX_N 105                                          // increase this if needed
struct Matrix { int mat[MAX_N][MAX_N]; };    // so that we can 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,
        ans.mat[i][j] += a.mat[i][k] * b.mat[k][j]; // do modulo arithmetic here
  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);    // check if p is odd (last bit is on)
    base = matMul(base, base);                                // square the base
    p >>= 1;                                                    // divide p by 2
  }
  return ans; }
```

Example code: ch5_07_UVa10681.cpp; ch5_07_UVa10681.java

Programming Exercises related to Powers of a (Square) Matrix:

1. **UVa 10229 - Modular Fibonacci *** (Fibonacci, matrix power, modulo arithmetic)
2. **UVa 10681 - Teobaldo's Trip *** (power of AdjMatrix, as discussed in this section)
3. **UVa 10870 - Recurrences *** (form the required matrix first; power of matrix)
4. UVa 12045 - Fun with Strings (eq with 2 unknowns, Fibonacci, matrix power, modulo)

5.10 Chapter Notes

Compared to the first edition of this book, this chapter has grown almost twice the size. However, we are aware that there are still many more mathematics problems and algorithms that have not been discussed in this chapter. Here, we list some pointers to more advanced topics that you may be interested to explore further by reading number theory books, e.g. [31], investigating mathematical topics in `http://mathworld.wolfram.com/` or Wikipedia, and do many more programming exercises related to mathematics problems like the ones in `http://projecteuler.net/` [8].

- There are many more **combinatorics** problems and formulas that are not yet discussed: **Burnside's lemma**, **Cayley's Formula**, **Derangement**, **Stirling Numbers**, etc.
- For an even **faster prime testing** function than the one presented here, one can use the non deterministic **Miller-Rabin's** algorithm – which can be made deterministic for contest environment with a known maximum input size N.
- In this chapter, we have seen a quite effective trial division method for finding prime factors of an integer and *lots of* its variant functions. For a **faster integer factorization**, one can use the **Pollard's rho** algorithm that uses another cycle detection algorithm called **Brent's cycle finding method**. However, if the integer to be factored is a large prime number, then this is still slow. This fact is the key idea of modern cryptography techniques.
- There are other theorems, hypothesis, and conjectures that cannot be discussed one by one, e.g. **Carmichael's function**, **Riemann's hypothesis**, **Goldbach's conjecture**, **Fermat's Little Test**, **twin prime conjecture**, **Chinese Remainder Theorem**, **Sprague-Grundy Theorem**, etc.
- To compute the solution of a **system of linear equations**, one can use techniques like **Gaussian elimination**.

As you can see, there are *many* topics about mathematics. This is not surprising since various mathematics problems have been investigated by people since many hundreds of 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 topics to be mastered is smaller, many IOI tasks require some form of 'mathematical insights'.

Note that (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.

There are $\approx$ **285 UVa (+ 10 others) programming exercises** discussed in this chapter.
(Only 175 in the first edition, a 69% increase).

There are **29 pages** in this chapter.
(Only 17 in the first edition, a 71% increase).

This page is intentionally left blank to keep the number of pages per chapter even.

Chapter 6

String Processing

The Human Genome has approximately 3.3 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 as graph and mathematics problems – namely: string processing. String processing is common in the research field of *bioinformatics*. However, as the strings that researchers deal with are usually extremely long, efficient data structures and algorithms were 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 [10] restriction. Additionally, the input and output format of IOI tasks are usually simple[1]. 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.

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 your favorite programming language (C, C++, 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 Appendix A). 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. Note: The sample input text file `'ch6.txt'` is shown on the next page; After question 1.(d) and before task 2.

[1]IOI 2010-2011 require contestants to implement function interfaces instead of coding I/O routines.

(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 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 looooooooooooooooong line...
```

2. Suppose 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 str = ''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 (like Knuth-Morris-Pratt (KMP) 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 lower/uppercase 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 (already in lowercase form), we will have these tokens = {'i', 'love', 'cs3233', 'competitive', 'programming', 'i', 'also', 'love', 'algorithm'}.

   (a) How to store an *array* of strings?
   (b) How to tokenize a string?

5. After that, we want to sort this array of strings lexicographically[2] and then find the lexicographically smallest string. That is, we want to have tokens sorted like this: {'algorithm', 'also', 'competitive', 'cs3233', 'i', 'i', 'love', 'love', 'programming'}. The answer for this example is 'algorithm'.

   (a) How to sort an array of strings lexicographically?

6. Now, identify which word appears the most in T. To do this, we need to count the frequency of each word. For T, the output is either 'i' or 'love', as both appear twice.

   (a) Which data structure best supports this word frequency counting problem?

7. The given text file has one more line after a line that starts with '.......'. The length of this last line is not constrained. Count how many characters are there in the last line?

   (a) How to read a string when we do not know its length in advance?

[2]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 it. 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 that purpose and many of the simpler ones end up as Ad Hoc programming contest problems. 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 simple form that can be dealt with modified input reader, i.e. C/C++ `scanf`, more complex problem involving some grammars[3] that requires recursive descent parser or Java Pattern class (Regular Expression)[4].

- Output Formatting
  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' for the contestants. Practice your coding skills by solving these problems *as fast as possible*.

- 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 (or have not been) classified as one of the other sub categories above.

Programming Exercises related to Ad Hoc String Processing:

- Cipher/Encode/Encrypt/Decode/Decrypt
  1. UVa 00213 - Message Decoding (decrypt the message)
  2. UVa 00245 - Uncompress (use the given algorithm)
  3. UVa 00306 - Cipher (can be made faster by avoiding cycle)
  4. UVa 00444 - Encoder and Decoder (each character is mapped to either 2 or 3 digits)
  5. UVa 00458 - The Decoder (shift each character's ASCII value by -7)
  6. UVa 00468 - Key to Success (letter frequency mapping)

[3]Interested readers should study more about Backus Naur Form (BNF) grammars.
[4]Some checking/counting problems can be solved with one liner code that use `matches()` or `replaceAll()` function.

7. UVa 00483 - Word Scramble (read char by char from left to right)
8. UVa 00492 - Pig Latin (ad hoc, similar to UVa 483)
9. UVa 00641 - Do the Untwist (reverse the given formula and simulate)
10. UVa 00739 - Soundex Indexing (straightforward conversion problem)
11. UVa 00740 - Baudot Data ... (just simulate the process)
12. UVa 00741 - Burrows Wheeler Decoder (simulate the process)
13. UVa 00795 - Sandorf's Cipher (prepare an 'inverse mapper')
14. UVa 00850 - Crypt Kicker II (plaintext attack)
15. UVa 00856 - The Vigenère Cipher (3 nested loops; one for each digit)
16. UVa 00865 - Substitution Cypher (simple character substitution mapping)
17. UVa 10222 - Decode the Mad Man (simple decoding mechanism)
18. **UVa 10851 - 2D Hieroglyphs ... *** (ignore border, '\/' = 1/0, read from bottom)
19. **UVa 10878 - Decode the Tape *** (treat space/'o' as 0/1, bin to dec conversion)
20. UVa 10896 - Known Plaintext Attack (complete search all possible keys; use tokenizer)
21. UVa 10921 - Find the Telephone (simple conversion problem)
22. UVa 11220 - Decoding the message (follow instruction in the problem)
23. UVa 11278 - One-Handed Typist (map QWERTY keys to DVORAK keys)
24. **UVa 11385 - Da Vinci Code *** (string manipulation + Fibonacci numbers)
25. UVa 11541 - Decoding (read char by char and simulate)
26. UVa 11697 - Playfair Cipher (follow the description, a bit tedious)
27. UVa 11716 - Digital Fortress (simple cipher)
28. UVa 11787 - Numeral Hieroglyphs (follow the description)

- Frequency Counting

1. UVa 00895 - Word Problem (get letter freq of each word, compare with puzzle line)
2. **UVa 00902 - Password Search *** (read char by char, count word frequency)
3. UVa 10008 - What's Cryptanalysis? (character frequency count)
4. UVa 10062 - Tell me the frequencies (ASCII character frequency count)
5. **UVa 10252 - Common Permutation *** (count the frequency of each alphabet)
6. UVa 10293 - Word Length and Frequency (straightforward)
7. UVa 10625 - GNU = GNU'sNotUnix (simulate the frequency addition n times)
8. UVa 10789 - Prime Frequency (report a letter if its frequency of appearance is prime)
9. **UVa 11203 - Can you decide it ... *** (problem desc hides this easy problem)
10. UVa 11577 - Letter Frequency (straightforward problem)

- Input Parsing

1. UVa 00271 - Simply Syntax (grammar check, linear scan)
2. UVa 00325 - Identifying Legal Pascal ... (follow the rule mentioned in description)
3. UVa 00327 - Evaluating Simple C ... (implementation can be tricky)
4. UVa 00384 - Slurpys (recursive grammar check)
5. UVa 00391 - Mark-up (use flags, tedious parsing)
6. UVa 00397 - Equation Elation (iteratively perform the next operation)
7. UVa 00442 - Matrix Chain Multiplication (parsing; properties of matrix chain mult)
8. UVa 00464 - Sentence/Phrase Generator (generate output based on BNF grammar)
9. UVa 00486 - English-Number Translator (recursive parsing)
10. UVa 00537 - Artificial Intelligence? (compute simple formula; parsing is difficult)
11. UVa 00576 - Haiku Review (parsing, grammar)
12. UVa 00620 - Cellular Structure (recursive grammar check)
13. **UVa 00622 - Grammar Evaluation *** (recursive BNF grammar check/evaluation)
14. UVa 00743 - The MTM Machine (recursive grammar check)
15. UVa 00933 - Water Flow (this simulation problem has complex input format)
16. **UVa 10058 - Jimmi's Riddles *** (solvable with Java regular expression)
17. UVa 10854 - Number of Paths (recursive parsing plus counting)

18. UVa 11148 - Moliu Fractions (extract ints, simple/mixed fractions from a line, gcd)
19. **UVa 11878 - Homework Checker *** (simple mathematical expression parsing)

- Output Formatting
  1. UVa 00320 - Border (requires flood fill technique)
  2. UVa 00445 - Marvelous Mazes (simulation, output formatting)
  3. **UVa 00488 - Triangle Wave *** (use several loops)
  4. UVa 00490 - Rotating Sentences (2d array manipulation, output formatting)
  5. UVa 10500 - Robot maps (simulate, output formatting)
  6. **UVa 10800 - Not That Kind of Graph *** (tedious output formatting problem)
  7. **UVa 10894 - Save Hridoy *** (how fast can you can solve this 'easy' problem?)
  8. UVa 11074 - Draw Grid (output formatting)
  9. UVa 11965 - Extra Spaces (replace consecutive spaces with only one space)
- String Comparison
  1. UVa 00409 - Excuses, Excuses (tokenize and compare; see if a word is an excuse word)
  2. **UVa 00644 - Immediate Decodability *** (is a code prefix to another?; brute force)
  3. UVa 00671 - Spell Checker (string comparison)
  4. **UVa 11048 - Automatic Correction ... *** (flexible string comp w.r.t dictionary)
  5. **UVa 11056 - Formula 1 *** (sorting and requires case-insensitive string comparison)
  6. UVa 11233 - Deli Deli (string comparison)
  7. UVa 11713 - Abstract Names (modified string comparison)
  8. UVa 11734 - Big Number of Teams will ... (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, simulation)
  3. UVa 00789 - Indexing (tokenizer, data structure; but no test data yet...)
  4. UVa 00892 - Finding words (basic string processing problem)
  5. **UVa 00941 - Permutations *** (find formula to get the n-th permutation of a string)
  6. UVa 10115 - Automatic Editing (simply do what they want, uses string)
  7. UVa 10197 - Learning Portuguese (must follow the description very closely)
  8. UVa 10361 - Automatic Poetry (read, tokenize, process as requested)
  9. UVa 10391 - Compound Words (more like data structure problem)
  10. **UVa 10393 - The One-Handed Typist *** (just follow the problem description)
  11. UVa 10508 - Word Morphing (key hint: number of words = number of letters + 1)
  12. UVa 10679 - I Love Strings (test data weak; checking if T is a prefix of S is AC)
  13. UVa 10761 - Broken Keyboard (tricky with output formatting; 'END' is part of input!)
  14. **UVa 11452 - Dancing the Cheeky-Cheeky *** (string period, small input, BF)
  15. UVa 11483 - Code Creator (straightforward, use 'escape character')
  16. UVa 11839 - Optical Reader (illegal if mark 0 or > 1 alternatives)
  17. UVa 11962 - DNA II (find formula; similar to UVa 941; base 4)
  18. LA 2972 - A DP Problem (Tehran03, tokenize linear equation)
  19. LA 3669 - String Cutting (Hanoi06)
  20. LA 3791 - Team Arrangement (Tehran06)
  21. LA 4144 - Greatest K-Palindrome ... (Jakarta08)
  22. LA 4200 - Find the Format String (Dhaka08)

Although the problems listed in this section constitute $\approx 77\%$ 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. In the next few sections, we will see string matching problems (Section 6.4) continued with string processing problems solvable with Dynamic Programming (DP) (Section 6.5). This chapter is closed with an extensive discussion on string processing problems where we have to deal with reasonably **long** strings, thus an efficient data structure for strings like Suffix **Trie**, Suffix **Tree**, or Suffix **Array** must be used. We discuss these data structures and several specialized algorithms using these data structures in Section 6.6.

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`). For 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 Solution

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 <cstring>, `find` in C++ <string>, `indexOf` in Java `String` class. We shall not elaborate further and let you explore these library functions by solving some of the programming exercises mentioned at the end of this section.

6.4.2 Knuth-Morris-Pratt (KMP) Algorithm

In Section 1.2.2, 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 (P[j] != T[i + j])                                 // if mismatch found
        found = false;               // abort this, shift 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.

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 `T[i + j]` and `P[j]` from `i` = 0 to 13 with `j = 0` (first character) is no different than the naïve algorithm.

```
              1         2         3         4         5
    012345678901234567890123456789012345678901234567890
T = I DO NOT LIKE SEVENTY SEV BUT SEVENTY SEVENTY SEVEN
P = SEVENTY SEVEN
    0123456789012
              1
    ^ the first character of P always mismatch with T[i] from index i = 0 to 13
    KMP has no choice but to shift starting index i by +1, as with naive matching.
```

[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.

```
... at i = 14 and j = 0 ...
              1         2         3         4         5
    012345678901234567890123456789012345678901234567890
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 algorithm will inefficiently restart from index i = 15 but KMP can resume from i = 25. This is because the match characters before mismatch is 'SEVENTY SEV'. 'SEV' (of length 3) appears as BOTH proper suffix and prefix of 'SEVENTY SEV'. This 'SEV' is also called as the **border** of 'SEVENTY SEV'. We can safely skip 'SEVENTY ' in 'SEVENTY 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 space), while i is still 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
    012345678901234567890123456789012345678901234567890
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
    012345678901234567890123456789012345678901234567890
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 'SEVENTY SEVEN' has 'SEVEN' (of length 5) as border, so KMP resets j back to 5, effectively skipping 13 - 5 = 8 characters of 'SEVENTY ' (notice the space again), immediately resumes the search from i = 43, and gets another match. This is efficient.

```
... at i = 43 and j = 5, then we have further matches from i = 43 to i = 50 ...
So P = 'SEVENTY SEVEN' is found again at index i = 38.
              1         2         3         4         5
    012345678901234567890123456789012345678901234567890
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
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
```

This means, if mismatch happens in `j = 11` (see 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.

```
#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];    // if different, reset j using b
    i++; j++;                                    // if same, advance both pointers
    b[i] = j;             // observe i = 8, 9, 10, 11, 12 with j = 0, 1, 2, 3, 4
} }                                 // 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];    // if 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
} } }
```

Exercise 6.4.1: Run `kmpPreprocess()` on `P = 'ABABA'`. What is the resulting reset table `b`?

Exercise 6.4.2: Run `kmpSearch()` to search pattern `P = 'ABABA'` in text `T = 'ACABAABABDABABA'`. Explain how the KMP search looks like?

Example codes: `ch6_01_kmp.cpp`; `ch6_01_kmp.java`

6.4.3 String Matching in a 2D Grid

A problem of string matching 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 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 the pattern 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 3.2.1). 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 choices* 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 8.3.2).

Programming Exercises related to String Matching

- Standard
  1. UVa 00455 - Periodic String (find s in s + s)
  2. **UVa 10298 - Power Strings *** (find s in s + s, similar to UVa 455)
  3. UVa 11362 - Phone List (string sort, matching)
  4. **UVa 11475 - Extend to Palindromes *** (investigate the term 'border' of KMP)
  5. **UVa 11576 - Scrolling Sign *** (modified string matching; complete search)
  6. UVa 11888 - Abnormal 89's (to check 'alindrome', find reverse of s in s + s)
- In 2D Grid
  1. **UVa 00422 - Word Search Wonder *** (2D grid, backtracking)
  2. **UVa 10010 - Where's Waldorf? *** (discussed in this section)
  3. **UVa 11283 - Playing Boggle *** (2D grid, backtracking, do not count twice)

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 Smith) is the basis for many sequence comparison programs.

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.

6.5 String Processing with Dynamic Programming

In this section, we discuss three string processing problems that are solvable with DP technique discussed in Section 3.5. The first two are *classical* problems and should be known by all competitive programmers. Additionally, we have added a collection of known twists of these problems.

Note that 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: Given two strings `A` and `B`, align[8] `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 (for now, 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), or
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 typically end up with a TLE verdict even for medium-length strings `A` and/or `B`. The solution for this problem is the well-known Needleman-Wunsch's DP algorithm [36]. 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[0..i]` to make the alignment
$V(0, j) = j \times score(\_, B[j])$ // insert spaces in `B[0..j]` to make the alignment

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).

```
A = 'xxx...xx'     A = 'xxx...xx'     A = 'xxx...x_'
            |                  |                  |
B = 'yyy...yy'     B = 'yyy...y_'     B = 'yyy...yy'
match/mismatch     delete             insert
```

[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'.

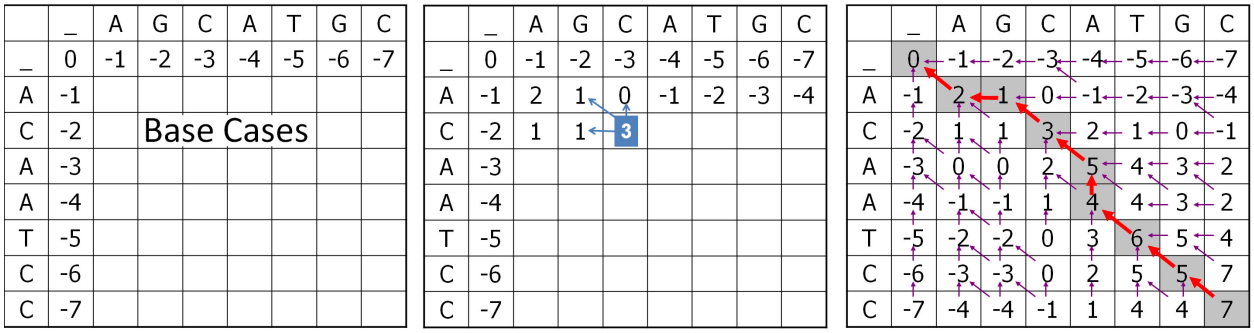


Figure 6.1: String Alignment Example for `A = 'ACAATCC'` and `B = 'AGCATGC'` (score = 7)

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 '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 DP algorithm is $O(nm)$ – the size of the DP table. We need to fill in all entries in the table. As each entry can be computed in $O(1)$, the time complexity is $O(nm)$.

Example codes: `ch6_02_str_align.cpp`; `ch6_02_str_align.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 for yourself.

Exercise 6.5.1.2: The example source codes: `ch6_02_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: Can we use the 'space saving trick' in Section 3.5? 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 discussed 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 (the **Smith-Waterman's** algorithm [36]) 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 string, can also be reduced to String Alignment problem, albeit in a non natural way. 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. For example, if you are given two permutations as in UVa 10635. Show how to solve this variant!

6.5.3 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, as shown in this example from UVa 11151 - Longest Palindrome: given a string of up to 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')

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 $len(l, r) = 1$. // true in odd-length palindrome
If $(l + 1 = r)$, then $len(l, r) = 2$ if $(A[l] = A[r])$, or 1 otherwise. // true in 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

---

Programming Exercises related to String Processing with DP:

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 + printing solution)
4. UVa 00963 - Spelling Corrector (this problem is problematic[9])
5. UVa 10066 - The Twin Towers (Longest Common Subsequence - but not on 'string')
6. UVa 10100 - Longest Match (Longest Common Subsequence)
7. UVa 10192 - Vacation (Longest Common Subsequence)
8. UVa 10405 - Longest Common Subsequence (as the problem name implies)
9. UVa 10617 - Again Palindrome (manipulate indices, not the actual string)
10. **UVa 10635 - Prince and Princess *** (find LCS of two permutations)
11. UVa 10739 - String to Palindrome (variation of edit distance)
12. **UVa 11151 - Longest Palindrome *** (discussed in this section)
13. UVa 11258 - String Partition (dp(i) = integer from substring `[i..k]` + dp(k))
14. LA 2460 - Searching Sequence ... (Singapore01, classical String Alignment problem)
15. LA 3170 - AGTC (Manila06, classical String Edit problem)

---

[9]Actually submitting a blank program that does nothing will get Accepted because there is no test data for this problem yet as of 1 August 2011. Anyway, this is a problem that uses Levenshtein distance.

6.6 Suffix Trie/Tree/Array

Suffix Trie, Suffix Tree, and Suffix Array are efficient and related data structures for strings. We did not discuss this topic in Section 2.3 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 a certain 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**[10] 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 possible 26 vertices (assuming that we only use uppercase Latin alphabets) according to the suffixes of strings in S. The common prefix of suffixes are 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'}. The Suffix Trie looks like Figure 6.2, with 7 suffix terminating vertices and 3 word terminating vertices.

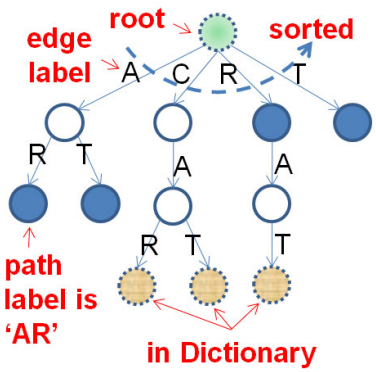


Figure 6.2: Suffix Trie

The typical purpose of Suffix Trie is as an efficient data structure for *dictionary*. For example, in Figure 6.2, if we want to check if the word 'CAT' exists in the dictionary, we can start from the root node, follow the edge with label 'C', then 'A', then 'T'. Since the vertex at this point has flag set to true, then we know that there is a word 'CAT' in the dictionary. Whereas, if we search for 'CAD', we go through root → 'C' → 'A' but then we do not have edge with edge label 'D', so we say that 'CAD' is not in the dictionary. Such dictionary queries are done in $O(m)$ where m is the length of the query/pattern string P – this is efficient[11].

6.6.2 Suffix Tree

i	Suffix
0	GATAGACA
1	ATAGACA
2	TAGACA
3	AGACA
4	GACA
5	ACA
6	CA
7	A

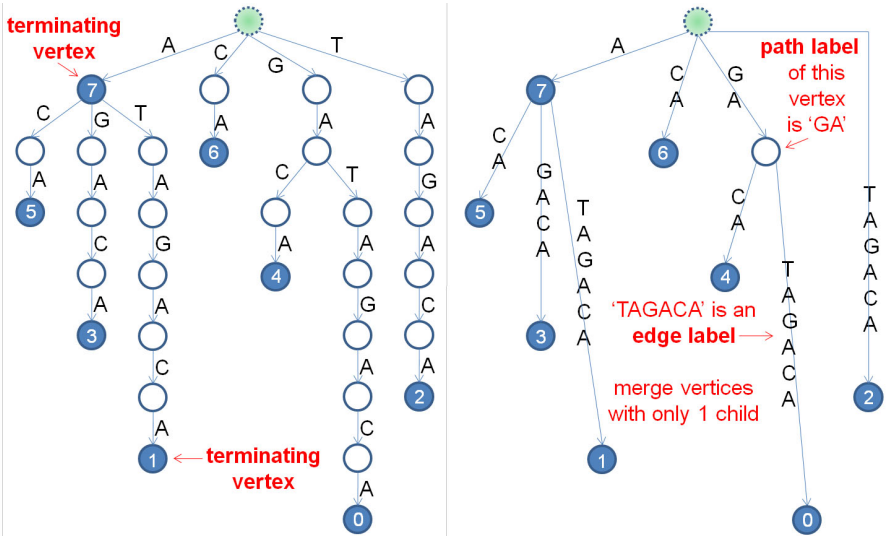


Figure 6.3: Suffixes, Suffix Trie, and Suffix Tree of `T = 'GATAGACA'`

[10] This is not a typo. The word 'TRIE' comes from the word 'information reTRIEval'.

[11] Another data structure for dictionary is balanced BST. 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 cost $O(m)$.

Now, instead of working with several short strings, we work with one *long(er)* string. Consider a string `T = 'GATAGACA'`, a 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. Most suffixes terminate at the leaf, but not all[12], for example suffix 7: 'A' in Figure 6.3. Observe that the longer the string `T` is, there will be more replicated 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. Suffix **Tree** is much more *compact* than Suffix **Trie** with at most $2n$ vertices only[13] (and thus at most $2n - 1$ edges).

Exercise 6.6.2.1: Draw the Suffix Trie and the Suffix Tree of `T = 'COMPETITIVE'`!

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:

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 many other 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 of the subtree rooted at x are the occurrences of `P` in `T`.

For 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[14] of 'A' in the subtree rooted at x. They are – sorted by the suffixes – suffix 7: `'A'`, suffix 5: `'ACA'`, suffix 3: `'AGACA'`, and suffix 1: `'ATAGACA'`.

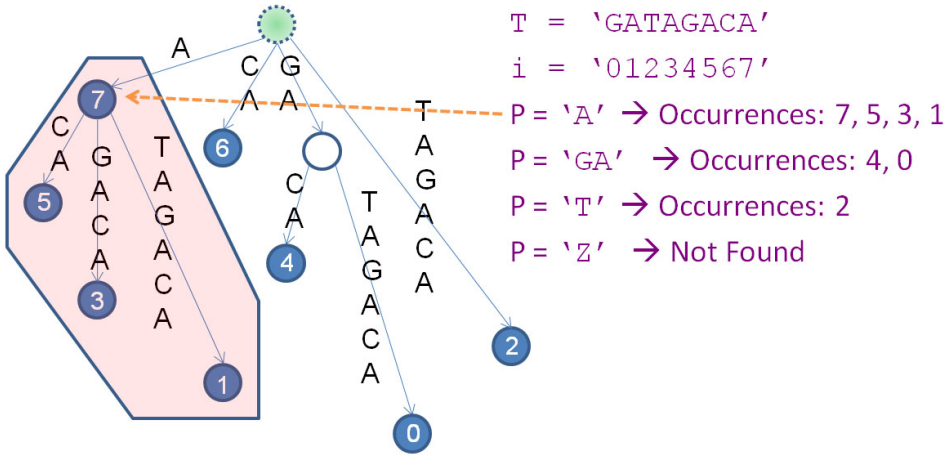


Figure 6.4: String Matching of `T = 'GATAGACA'` with Various Pattern Strings

Exercise 6.6.3.1: Given the same Suffix Tree in Figure 6.4, find `P = 'CA'` and `P = 'CAT'`!

[12]Other Suffix Tree implementation uses a special *terminating symbol* like the dollar ('$') sign. If a terminating symbol is used, it will *always* appear at the end of all n suffixes and at all n leaves. However, in Competitive Programming, we usually do not need this, especially later in our Suffix Array implementation.

[13]There can be 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.

[14]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 in the subtree rooted at x.

Longest Repeated Substring in $O(n)$

Given the Suffix Tree of `T`, we can also find the Longest Repeated Substring[15] (LRS) in T efficiently. 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 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.

For example, in the Suffix Tree of `T = 'GATAGACA'` shown in Figure 6.5, the LRS is `'GA'` as it is the path label of the deepest internal vertex x. See that `'GA'` is repeated twice in `'GATAGACA'`.

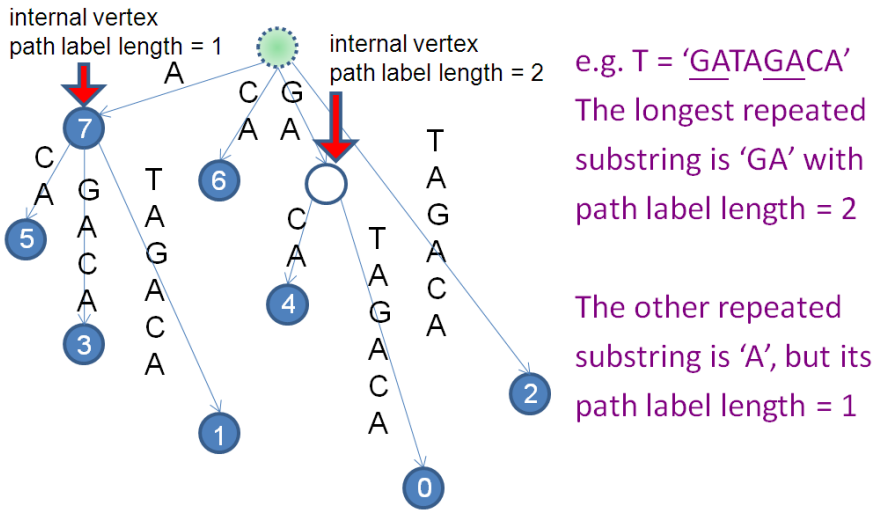


Figure 6.5: Longest Repeated Substring of `T = 'GATAGACA'`

Exercise 6.6.3.2: Find the LRS in `T = 'CGACATTACATTA'`! Build the Suffix Tree first.

Longest Common Substring in $O(n)$

The problem of finding the Longest Common **Substring** (not 'Subsequence')[16] of two **or more** strings can be solved in linear time[17] 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. 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.

Exercise 6.6.3.3: Find the Longest Common Substring of T_1 `= 'STEVEN'` and T_2 `= 'SEVEN'`!

Exercise 6.6.3.4: Think of how to generalize this approach to find the LCS of *more than two strings*. Example: Find the LCS between T_1 `= 'STEVEN'`, T_2 `= 'SEVEN'`, and T_3 `= 'EVE'`!

Exercise 6.6.3.5: Customize the solution further so that we find the LCS of k *out of* n *strings*, where $k \leq n$. Example: Using T_1, T_2, and T_3 above, find the LCS of 2 out of 3 strings.

[15]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.

[16]In 'ABCDEF', 'BCE' is a subsequence but not a substring whereas 'BCD' (contiguous) is both.

[17]Only if we use the linear time Suffix Tree construction algorithm (not discussed in this book, see [37]).

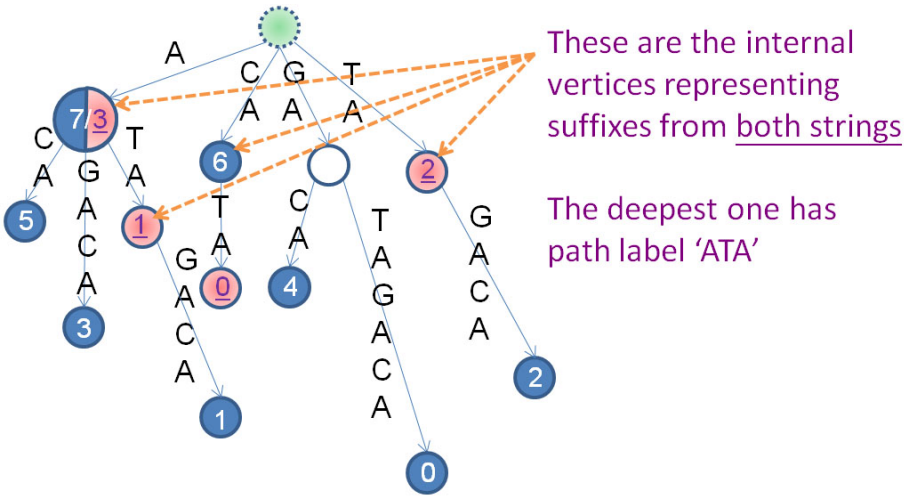


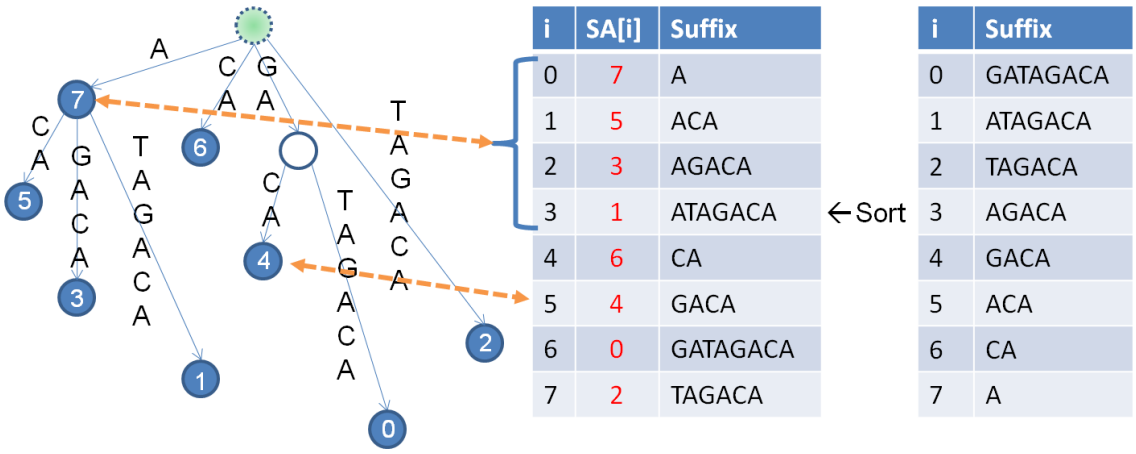
Figure 6.6: Generalized Suffix Tree of T_1 = 'GATAGACA' and T_2 = 'CATA' and their LCS

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 is complex and thus risky under programming contest setting. Fortunately, *another* data structure that we are going to describe – the **Suffix Array** invented by Udi Manber and Gene Myers – 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 [37] and instead focus on $O(n \log n)$ Suffix Array construction [40] which is easier to use. Then, in the next subsection, we will show that we can apply Suffix Array to solve problems that are shown to be solvable with Suffix Tree.

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 $n = 8$. The Suffix Array of `T` is a permutation of integers `[0..n-1] = {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 `7` = `'A'`, suffix `SA[1]` = suffix `5` = `'ACA'`, ..., and finally suffix `SA[7]` = suffix `2` = `'TAGACA'`.

Suffix Tree and Suffix Array are closely related. As we can see in Figure 6.7, the preorder traversal of the Suffix Tree visits the terminating vertices 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** (at leaf) 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.



i	SA[i]	Suffix
0	7	A
1	5	ACA
2	3	AGACA
3	1	ATAGACA
4	6	CA
5	4	GACA
6	0	GATAGACA
7	2	TAGACA

←Sort

i	Suffix
0	GATAGACA
1	ATAGACA
2	TAGACA
3	AGACA
4	GACA
5	ACA
6	CA
7	A

Figure 6.7: Suffix Tree and Suffix Array of `T = 'GATAGACA'`

Suffix Array is good enough for many challenging string operations 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 characters
int SA[MAX_N], i, n;

bool cmp(int a, int b) { return strcmp(T + a, T + b) < 0; }  // compare suffixes

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) * comparison: 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 sorting and string comparison *library* produces the correct Suffix Array = {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 suffix comparison.

A *better way* to construct Suffix Array is to sort the *ranking pairs* (small integers) of the suffixes in $O(\log_2 n)$ iterations from $k = 1, 2, 4, \ldots$, last power of 2 that is less than n. At each iteration, this construction algorithm sort 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 [40]. An example of how this algorithm works is shown below for `T = 'GATAGACA'`.

- Initially, `SA[i] = i` and `RA[i] = T[i]-'.'` for all `i` ∈ `[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])`. Initially, this ranking pair is obtained by subtracting the ASCII value of the first and the second characters of that suffix with the ASCII value of '.' (This '.' will be used as a delimiter character for the Longest Common Substring problem). Example 1: The rank of suffix 5 `'ACA'` is ('A'-'.', 'C'-'.') = (19, 21). Example 2: The rank of suffix 3 `'AGACA'` is ('A'-'.', 'G'-'.') = (19, 25). After we sort these ranking pairs, the order of suffixes is now like Table 6.1, right, with suffix 5 `'ACA'` before suffix 3 `'AGACA'`, etc.

i	SA[i]	Suffix	RA[SA[i]]	RA[SA[i]+1]
0	0	GATAGACA	25 (G)	19 (A)
1	1	ATAGACA	19 (A)	38 (T)
2	2	TAGACA	38 (T)	19 (A)
3	3	AGACA	19 (A)	25 (G)
4	4	GACA	25 (G)	19 (A)
5	5	ACA	19 (A)	21 (C)
6	6	CA	21 (C)	19 (A)
7	7	A	19 (A)	00 ($)

Initial ranks RA[i] = T[i] - ASCII '.'
A = 19, C = 21, G = 25, T = 38

i	SA[i]	Suffix	RA[SA[i]]	RA[SA[i]+1]
0	7	A	19 (A)	00 ($)
1	5	ACA	19 (A)	21 (C)
2	3	AGACA	19 (A)	25 (G)
3	1	ATAGACA	19 (A)	38 (T)
4	6	CA	21 (C)	19 (A)
5	0	GATAGACA	25 (G)	19 (A)
6	4	GACA	25 (G)	19 (A)
7	2	TAGACA	38 (T)	19 (A)

If SA[i] + k >= n (beyond the length of string T),
we give a default rank 0 with label $

Table 6.1: Left/Right: Before/After Sorting; k = 1; Initial Sorted Order Appears

- 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 7 `'A'` 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.

  Example 1: The ranking pair of suffix 5 `'ACA'` is (19, 21). This is different with the ranking pair of previous suffix 7 `'A'` which is (19, 00). So suffix 5 has a new rank 0 + 1 = 1.

  Example 2: The ranking pair of suffix 4 `'GACA'` is (25, 19). This is similar with the ranking pair of previous suffix 0 `'GATAGACA'` which is also (25, 19). Since suffix 0 is given new rank 5, then suffix 4 is also given the same new rank 5.

  Now, the new ranking pair of suffix 7 `'A'` is (0, 0) and suffix 5 `'ACA'` is (1, 0). These two suffixes are still in sorted order. However, the ranking pair of suffix 0 `'GATAGACA'` is (5, 6) and suffix 4 `'GACA'` is (5, 4). These two suffixes are still not in sorted order. After another round of sorting, the order of suffixes is now like Table 6.2, right.

i	SA[i]	Suffix	RA[SA[i]]	RA[SA[i]+2]
0	7	A	0 (A$)	0 ($$)
1	5	ACA	1 (AC)	0 (A$)
2	3	AGACA	2 (AG)	1 (AC)
3	1	ATAGACA	3 (AT)	2 (AG)
4	6	CA	4 (CA)	0 ($$)
5	0	GATAGACA	5 (GA)	6 (TA)
6	4	GACA	5 (GA)	4 (CA)
7	2	TAGACA	6 (TA)	5 (GA)

A$ (first item) is given rank 0, then for i = 1 to n-1, compare previous ranks row by row

i	SA[i]	Suffix	RA[SA[i]]	RA[SA[i]+2]
0	7	A	0 (A$)	0 ($$)
1	5	ACA	1 (AC)	0 (A$)
2	3	AGACA	2 (AG)	1 (AC)
3	1	ATAGACA	3 (AT)	2 (AG)
4	6	CA	4 (CA)	0 ($$)
5	4	GACA	5 (GA)	4 (CA)
6	0	GATAGACA	5 (GA)	6 (TA)
7	2	TAGACA	6 (TA)	5 (GA)

If SA[i] + k >= n (beyond the length of string T), we give a default rank 0 with label $

Table 6.2: Left/Right: Before/After Sorting; k = 2; `'GATAGACA'` and `'GACA'` are Swapped

- 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 (5, 4) and Suffix 0 (5, 6) are now different, so after re-ranking, all n suffixes now have different ranking. When this happens, we have successfully obtained the Suffix Array (see Table 6.3).

i	SA[i]	Suffix	RA[SA[i]]	RA[SA[i]+4]
0	7	A	0 (A$$$)	0 ($$$$)
1	5	ACA	1 (ACA$)	0 ($$$$)
2	3	AGACA	2 (AGAC)	0 (A$$$)
3	1	ATAGACA	3 (ATAG)	0 (ACA$)
4	6	CA	4 (CA$$)	0 ($$$$)
5	4	GACA	5 (GACA)	0 ($$$$)
6	0	GATAGACA	6 (GATA)	5 (GACA)
7	2	TAGACA	7 (TAGA)	0 (CA$$)

A$$$$ (first item) is given rank 0, all other suffices also have different previous ranking pairs → done

Table 6.3: Before and After sorting; k = 4; No Change

We can implement the sorting of ranking pairs above using (built-in) $O(n \log n)$ sorting library. As we repeat the sorting process $\log n$ times, the overall time complexity is $O(\log n \times n \log n) = O(n \log^2 n)$. With this 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 – details omitted) to reduce the sorting time to $O(n)$. As we repeat sorting $\log n$ 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. For both ICPC and IOI contestants, 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.

```
#include <algorithm>
#include <cstdio>
#include <cstring>
using namespace std;

#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) {
  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 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];
} }

int main() {
  n = (int)strlen(gets(T));
  constructSA();
  for (int i = 0; i < n; i++) printf("%2d\t%s\n", SA[i], T + SA[i]);
} // return 0;
```

Example codes: ch6_03_sa.cpp; ch6_03_sa.java

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 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[18]. 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/upper bound as the first/second item of the pair, respectively

int main() {
  n = (int)strlen(gets(T));
  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 when an empty string is entered
    ii pos = stringMatching();
    if (pos.first != -1 && pos.second != -1) {
      printf("%s is 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;
```

[18] This is achievable by using the `strncmp` function to compare only the first m characters of both suffixes.

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..7]` and thus the middle one is `i = 3`. We compare the first two characters of suffix `SA[3]`, which is 'ATAGACA', with `P = 'GA'`. As `P = 'GA'` is larger, we continue exploring `i = [4..7]`. Next, we compare the first two characters of suffix `SA[5]`, which is 'GACA', with `P = 'GA'`. It is a match. As we are currently looking for the *lower* bound, we do not stop here but continue exploring `i = [4..5]`. `P = 'GA'` is larger than suffix `SA[4]`, which is 'CA', so we continue exploring `i = [5..5]`. We only have one item, so we stop here. Index `i = 5` is the lower bound, i.e. suffix `SA[5]`, which is 'GACA', is the *first* time pattern `P = 'GA'` appears as a prefix of a suffix in the list of sorted suffixes.

Next, we search for the upper bound. The first step is similar. But at the second step, we have a match between suffix `SA[5]`, which is 'GACA', with `P = 'GA'`. Since now we are looking for the *upper* bound, we continue exploring `i = [6..7]`. We found another match when comparing suffix `SA[6]`, which is 'GATAGACA', with `P = 'GA'`. This `i = 6` is the upper bound in this example, i.e. suffix `SA[6]`, which is 'GATAGACA', is the *last* time pattern `P = 'GA'` appears as a prefix of a suffix in the list of sorted suffixes.

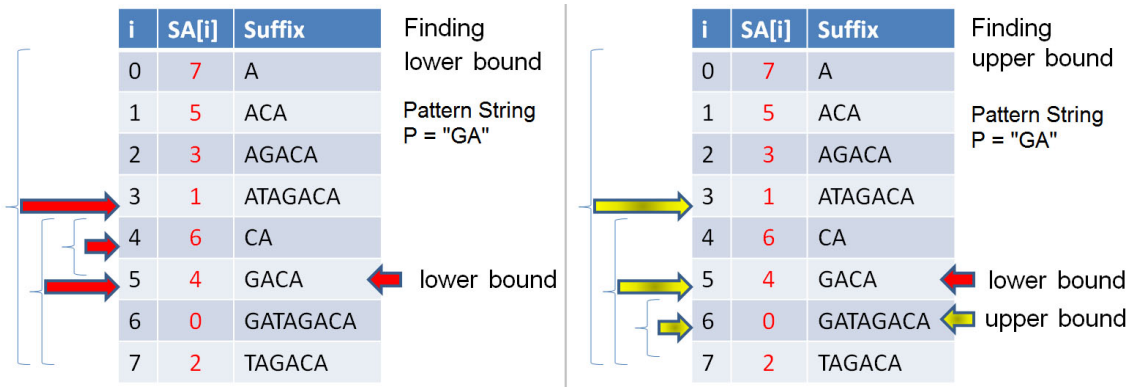


Table 6.4: String Matching using Suffix Array

Longest Common Prefix in $O(n)$

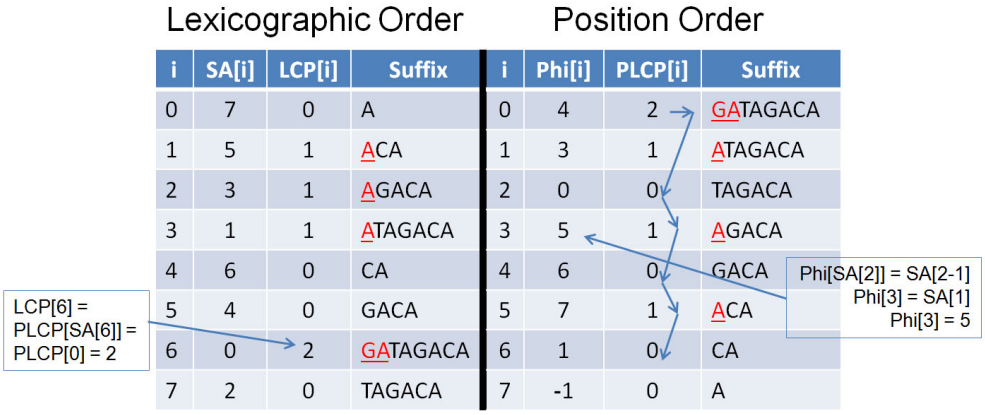
Given the Suffix Array of `T`, we can compute the Longest Common Prefix (LCP) between *consecutive* suffixes in Suffix Array order. See Table 6.5, left side. By definition, `LCP[0] = 0` as suffix `SA[0]` is the first suffix in Suffix Array order without any other suffix preceding it. For `i > 0, LCP[i]` = length of common prefix between suffix `SA[i]` and suffix `SA[i-1]`. 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.

```
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, increase L
    LCP[i] = L;
} }
```

A better idea using the Permuted Longest-Common-Prefix (PLCP) theorem [22] 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 ($1 \rightarrow 0$).

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[2]] = SA[2-1]`, so `Phi[3] = SA[1] = 5`.



i	SA[i]	LCP[i]	Suffix	i	Phi[i]	PLCP[i]	Suffix
0	7	0	A	0	4	2	GATAGACA
1	5	1	ACA	1	3	1	ATAGACA
2	3	1	AGACA	2	0	0	TAGACA
3	1	1	ATAGACA	3	5	1	AGACA
4	6	0	CA	4	6	0	GACA
5	4	0	GACA	5	7	1	ACA
6	0	2	GATAGACA	6	1	0	CA
7	2	0	TAGACA	7	-1	0	A

Table 6.5: Computing the Longest Common Prefix (LCP) given the SA of `T = 'GATAGACA'`

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. 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 $O(n)$. 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 will be increased max n times
    PLCP[i] = L;
    L = max(L-1, 0);                          // L will be decreased max n times
  }
  for (i = 1; i < n; i++)                                 // compute LCP in O(n)
    LCP[i] = PLCP[SA[i]];   // put the permuted LCP back to the correct position
}
```

Longest Repeated Substring in $O(n)$

If we have computed the Suffix Array and the LCP between consecutive suffixes in Suffix Array order, 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 = 'GATAGACA'`, the highest number is 2 at index `i = 6`. The first 2 characters of the corresponding suffix `SA[6]` (suffix `0`) is 'GA'. This is the longest repeated substring in `T`.

Longest Common Substring in $O(n)$

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 = `'GATAGACA'` and T_2 = `'CATA'`. To solve the LCS problem using Suffix Array, first we have to concatenate both strings with a delimiter that does not appear in either string. We use a period[19] '.' to produce `T = 'GATAGACA.CATA'`. Then, we compute the Suffix and LCP array of `T` as shown in Figure 6.8.

Then, we go through consecutive suffixes in $O(n)$. If two consecutive suffixes belong to different owner (can be easily checked[20] 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 6.8, this happens when `i = 6`, as suffix `SA[6]` = suffix `1` = `'ATAGACA.CATA'` (owned by T_1) and its previous suffix `SA[5]` = suffix `10` = `'ATA'` (owned by T_2) have a common prefix of length 3 which is `'ATA'`.

i	SA[i]	LCP[i]	Owner	Suffix
0	8	0	2/NA	.CATA
1	12	0	2	A
2	7	1	1	A.CATA
3	5	1	1	ACA.CATA
4	3	1	1	AGACA.CATA
5	10	1	2	ATA
6	1	3	1	ATAGACA.CATA
7	6	0	1	CA.CATA
8	9	2	2	CATA
9	4	0	1	GACA.CATA
10	0	2	1	GATAGACA.CATA
11	11	0	2	TA
12	2	2	1	TAGACA.CATA

Figure 6.8: The Suffix Array, LCP, and owner of `T = 'GATAGACA.CATA'`

Programming Exercises related to Suffix Array[21]:

1. UVa 00719 - Glass Beads (min lexicographic rotation[22], $O(n \log n)$ SA construction)
2. **UVa 00760 - DNA Sequencing *** (Longest Common *Substring* between two strings)
3. **UVa 11107 - Life Forms *** (Longest Common Substring between $> \frac{1}{2}$ of the strings)
4. **UVa 11512 - GATTACA *** (Longest Repeated Substring)
5. LA 3901 - Editor (Seoul07, Longest Repeated Substring (or KMP))
6. LA 3999 - The longest constant gene (Danang07, LC Substring of ≥ 2 strings)
7. LA 4657 - Top 10 (Jakarta09, Suffix Array + Segment Tree, problem author: Felix Halim)
8. SPOJ 6409 - Suffix Array (problem author: Felix Halim)
9. IOI 2008 - Type Printer (DFS traversal of Suffix Trie)

Exercise 6.6.5.1: Find the LRS in `T = 'CGACATTACATTA'`! Build the Suffix Array first.

Exercise 6.6.5.2: Find the Longest Common Substring of T_1 = `'STEVEN'` and T_2 = `'SEVEN'`! Build the Suffix Array of the combined string first.

[19] The period '.' is the ASCII value used for giving the initial ranking of `RA[SA[i]], RA[SA[i]+1]` in the $O(n \log n)$ Suffix Array construction code shown earlier. If you want to change this delimiter to another character, please make sure that you understand how the given Suffix Array construction code works.

[20] With three or more strings, this check will have more 'if statements'.

[21] 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.

[22] 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 Chapter Notes

The material about String Alignment (Edit Distance), Longest Common Subsequence, Suffix Tree, and Suffix Array are originally from **A/P Sung Wing Kin, Ken** [36], School of Computing, National University of Singapore. The materials from A/P Ken's lecture notes have since evolved from 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 are 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.

Due to some personal requests, we have decided to include a section on the String Matching problem (Section 6.4). We discussed the library solutions and one fast algorithm (Knuth-Morris-Pratt/KMP algorithm). 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 6.6. 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 the famous String Alignment (Edit Distance) problem and its related Longest Common Subsequence problem in Section 6.5. There are several interesting exercises that discuss the variants of these two problems.

The practical implementation of Suffix Array (Section 6.6) is inspired mainly from the article "Suffix arrays - a programming contest approach" by [40]. We have integrated and synchronized many examples given there with our way of writing Suffix Array implementation – a total overhaul compared with the version in the first edition. It is a good idea to solve *all* the programming exercises listed in that section although they are not that many *yet*. This is an important data structure that will be more and more popular in the near future.

Compared to the first edition of this book, this chapter has grown almost twice the size. 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** (more efficient Trie data structure), etc.

There are $\approx$ **117 UVa (+ 12 others) programming exercises** discussed in this chapter.
(only 54 in the first edition, a 138% increase).

There are **24 pages** in this chapter
(only 10 in the first edition, a 140% increase).

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 and 2010), IOI tasks do not feature geometry problems. However, in earlier years, there exists one or two geometry related problems per IOI [39].

We have observed that many contestants, especially ICPC contestants, are 'afraid' to tackle geometry-related problems due to two logical reasons:

1. 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, i.e. Complete Search or Dynamic Programming problems. This make attempting *other* problem types in the problem set more worthwhile than spending precious minutes coding geometry solution that has lower probability of acceptance.

    - There are usually several tricky 'special corner cases' in geometry problems, e.g. What if the lines are vertical (infinite gradient)?, What if the points are collinear?, What if the polygon is concave?, etc. It is usually a very good idea to test your team's geometry solution with lots of 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.

2. The contestants are not well prepared.

    - The contestants forget some important basic formulas or unable to derive the required formulas from the basic ones.
    - The contestants do not prepare well-written library functions and their attempts to code such functions during stressful contest time end up with (lots of) bugs. In ICPC, the top teams usually fill sizeable part of their hard copy material (which they can bring into the contest room) with lots of geometry formulas and library functions.

[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.

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.

In Section 7.2, we present many (it is impossible to enumerate all) English geometric terminologies[2] 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 not sure of certain terminologies or forget some basic formulas.

In Section 7.3, we discuss 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, finding the convex hull of a set of points, cutting a polygon with a straight line, etc.

We end this chapter by revisiting **Divide and Conquer** technique for geometry-related problems in Section 7.4. Our 2D (and 3D) world can be divided into sections/zones that are *not* overlapping, making Divide and Conquer a good approach for tackling geometry problems (as opposed to Dynamic Programming approach).

The implementation of the formulas and computational geometry algorithms shown in this chapter uses the following techniques to increase the probability of acceptance:

1. We highlight the special cases that can 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 like `1e-9` instead of testing `a == b`.

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[3] Euclidean[4] space, points are usually represented with a struct in C/C++ (or Class in Java) with two 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.
   A constructor can be used to (slightly) simplify coding later.
2. Sometimes we need to sort the points. We can easily do that with the trick below.

```
// 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(int _x, int _y) { x = _x, y = _y; }      // constructor (optional)
};
struct point { double x, y;         // only used if more precision is needed
  point(double _x, double _y) { x = _x, y = _y; }             // constructor
  bool operator < (point other) {           // override 'less than' operator
    if (fabs(x - other.x) < EPS)                        // useful for sorting
      return x < other.x;                      // first criteria , by x-axis
    return y < other.y;                        // second criteria, by y-axis
} };
```

[2]ACM ICPC and IOI contestants come from various nationalities and backgrounds. Therefore, we would like to get everyone familiarized with English geometric terminologies.

[3]Add one more member, `z`, if you are working in 3D Euclidean space.

[4]To make this simple, the 2D and 3D Euclidean spaces are the 2D and 3D world that we encounter in real life.

```
// in int main(), assuming we already have a populated vector<point> P;
  sort(P.begin(), P.end());          // comparison operator is defined above
```

3. We can easily test if two points are the same with the function below.
   Note the differences between integer and floating point versions.

```
bool areSame(point_i p1, point_i p2) {                    // integer version
  return p1.x == p2.x && p1.y == p2.y; }               // precise comparison

bool areSame(point p1, point p2) { // floating point version
  // use EPS when testing equality of two floating points
  return fabs(p1.x - p2.x) < EPS && fabs(p1.y - p2.y) < EPS; }
```

4. We can measure the Euclidean distance[5] 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
```

   Exercise 7.2.1.1: Compute the Euclidean distance between point (2.0, 2.0) and (6.0, 5.0)!

5. We can rotate a point by certain angle around origin by using a rotation matrix.

```
// rotate p by theta degrees CCW w.r.t origin (0, 0)
point rotate(point p, double theta) {
  // rotation matrix R(theta) = [cos(theta) -sin(theta)]
  //                            [sin(theta)  cos(theta)]
  // usage: [x'] = R(theta) * [x]
  //        [y']              [y]
  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 7.2.1.2: Rotate a point (4.0, 3.0) by 90 degrees counter clockwise around origin. What is the new coordinate of the rotated point? (easy to compute by hand).

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 this linear equation with $b = 1$ 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
```

   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?

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.

[5]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.

```
// the answer is stored in the third parameter (pass by reference)
void pointsToLine(point p1, point p2, line *l) {
  if (p1.x == p2.x) {                // vertical line is handled nicely here
    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;
    l->c = -(double)(l->a * p1.x) - (l->b * p1.y);
} }
```

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 actually compute the required line equation if we are only given *one* point, but we *also* need to know the gradient/slope of that line. Show how to compute line equation given a point and gradient!

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).

```
bool areParallel(line l1, line l2) {              // check coefficient 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[6] are not the same and are not parallel, they will intersect at a point. That intersection point (x, y) can be found by solving the system of two linear algebraic equations 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 (areSame(l1, l2)) return false;                // all points intersect
  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);
  if (fabs(l1.b) > EPS)              // special case: test for vertical line
    p->y = - (l1.a * p->x + l1.c) / l1.b;          // avoid division by zero
  else
    p->y = - (l2.a * p->x + l2.c) / l2.b;
  return true; }
```

5. **Line Segment** is a line with two end points with *finite length*.

6. **Vector**[7] is a line segment (thus it has two end points and length/magnitude) with a *direction*. Usually, 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.

[6]To avoid confusion, please differentiate between line intersection versus line *segment* intersection.
[7]Do not confuse this with C++ STL `vector`.

7. We can translate (move) a point w.r.t a vector as a vector describes the displacement magnitude in x and y-axis.

```
struct vec { double x, y;   // we use 'vec' to differentiate with STL vector
  vec(double _x, double _y) { x = _x, y = _y; } };

vec toVector(point p1, point p2) {              // convert 2 points to vector
  return vec(p2.x - p1.x, p2.y - p1.y); }

vec scaleVector(vec v, double s) {  // nonnegative s = [<1 ...   1   ... >1]
  return vec(v.x * s, v.y * s); }             // shorter v  same v  longer v

point translate(point p, vec v) {               // translate p according to v
  return point(p.x + v.x , p.y + v.y); }
```

Exercise 7.2.2.6: Translate a point c (3.0, 2.0) according to a vector $\overrightarrow{ab}$ defined by point a (2.0, 2.0) and b (4.0, 3.0).
What is the new coordinate of the translated point? (easy to compute by hand).

Exercise 7.2.2.7: Same as **Exercise 7.2.2.6** above, but now the magnitude of vector $\overrightarrow{ab}$ is reduced by *half*.
What is the new coordinate of the translated point? (still easy to compute by hand).

Exercise 7.2.2.8: Same as **Exercise 7.2.2.6** above, then immediately rotate the resulting point by 90 degrees counter clockwise around origin.
What is the new coordinate of the translated point? (still easy to compute by hand).

Exercise 7.2.2.9: Rotate a point c (3.0, 2.0) by 90 degrees counter clockwise around origin, then immediately translate the resulting point according to a vector $\overrightarrow{ab}$. Vector $\overrightarrow{ab}$ is the same as in **Exercise 7.2.2.6** above.
What is the new coordinate of the translated point? (still easy to compute by hand).
Is the result similar with the previous **Exercise 7.2.2.8** above?
What can we learn from this phenomenon?

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 7.1, 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 $\overrightarrow{ab}$, or $c = a + u \times \overrightarrow{ab}$. Point c is located at the intersection point between line l and another line l' *perpendicular* to l that pass through point p, thus the *dot product* of l and l' is 0. We can rework these equations to obtain c. See the implementation below.

   Exercise 7.2.2.10: We can also compute the location of point c by finding that other line l' that is perpendicular with line l that 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?

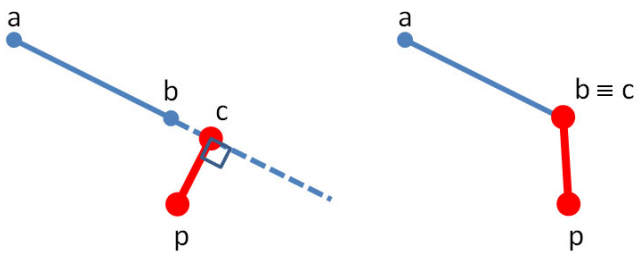


Figure 7.1: Distance to Line (left) and to Line Segment (right)

9. If we are given 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 7.1, right). See the implementation below.

```
// 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 (by reference)
double distToLine(point p, point A, point B, point *c) {
  // formula: cp = A + (p-A).(B-A) / |B-A| * (B-A)
  double scale = (double)
    ((p.x - A.x) * (B.x - A.x) + (p.y - A.y) * (B.y - A.y)) /
    ((B.x - A.x) * (B.x - A.x) + (B.y - A.y) * (B.y - A.y));
  c->x = A.x + scale * (B.x - A.x);
  c->y = A.y + scale * (B.y - A.y);
  return dist(p, *c); }               // Euclidean distance between p and *c

// returns the distance from p to the line segment ab (still OK if A == B)
// the closest point is stored in the 4th parameter (by reference)
double distToLineSegment(point p, point A, point B, point* c) {
  if ((B.x-A.x) * (p.x-A.x) + (B.y-A.y) * (p.y-A.y) < EPS) {
    c->x = A.x;   c->y = A.y;                                  // closer to A
    return dist(p, A); }               // Euclidean distance between p and A
  if ((A.x-B.x) * (p.x-B.x) + (A.y-B.y) * (p.y-B.y) < EPS) {
    c->x = B.x;   c->y = B.y;                                  // closer to B
    return dist(p, B); }               // Euclidean distance between p and B
  return distToLine(p, A, B, c); }              // call distToLine as above
```

10. Given a line defined by two points p and q, we can determine whether 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 $\overrightarrow{pq}$ and $\overrightarrow{pr}$ be the two vectors obtained from these three points. The cross product $\overrightarrow{pq} \times \overrightarrow{pr}$ result in another vector that is perpendicular to both $\overrightarrow{pq}$ and $\overrightarrow{pr}$. The magnitude of this vector is equal to the area of the *parallelogram* that the vectors span[8]. If the magnitude is positive/zero/negative, then we have a left turn/collinear/right turn, respectively. The left turn test is more famously known as the **CCW (Counter Clockwise) Test**.

```
double cross(point p, point q, point r) {                  // cross product
  return (r.x - q.x) * (p.y - q.y) - (r.y - q.y) * (p.x - q.x); }

// 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(p, q, r)) < EPS; }     // notice the comparison with EPS

// returns true if point r is on the left side of line pq
bool ccw(point p, point q, point r) {
  return cross(p, q, r) > 0; } // can be modified to accept collinear points
```

Example codes: ch7_01_points_lines.cpp; ch7_01_points_lines.java

[8]The area of triangle pqr is therefore *half* of the area of this parallelogram.

Programming Exercises related to Points and Lines:

1. UVa 00184 - Laser Lines (brute force; `collinear` test)
2. UVa 00191 - Intersection (line segment intersection)
3. UVa 00270 - Lining Up (gradient sorting, complete search)
4. UVa 00356 - Square Pegs And Round Holes (Euclidean distance, brute force)
5. UVa 00378 - Intersecting Lines (use `areParallel`, `areSame`, `areIntersect`)
6. UVa 00587 - There's treasure everywhere (Euclidean distance `dist`)
7. UVa 00833 - Water Falls (recursive check, use the `ccw` tests)
8. UVa 00837 - Light and Transparencies (line segments, sort the x-axis first)
9. **UVa 00920 - Sunny Mountains *** (Euclidean distance `dist`)
10. UVa 10167 - Birthday Cake (brute force A and B, `ccw` tests)
11. UVa 10242 - Fourth Point (use `toVector`, then `translate` points w.r.t that vector)
12. **UVa 10263 - Railway *** (use `distToLineSegment`)
13. UVa 10310 - Dog and Gopher (complete search, Euclidean distance `dist`)
14. UVa 10357 - Playball (Euclidean distance `dist`, simple Physics simulation)
15. UVa 10466 - How Far? (Euclidean distance `dist`)
16. UVa 10585 - Center of symmetry (sort the points)
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 eqs with 2 unknowns)
20. **UVa 11227 - The silver bullet *** (brute force; `collinear` test)
21. UVa 11343 - Isolated Segments (line segment intersection)
22. UVa 11505 - Logo (Euclidean distance `dist`)
23. LA 4601 - Euclid (Southeast USA Regional 2009)

7.2.3 2D Objects: Circles

1. **Circle** centered at (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 at the border of a circle, we can use the following function. Modify this function a bit for the floating point version.

```
int inCircle(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
```

3. The constant **Pi (π)** is the ratio of *any* circle's circumference to its diameter. To avoid unnecessary precision error, the safest value for programming contest if this constant π is not defined in the problem description is[9] `pi = acos(-1.0)` or `pi = 2 * acos (0.0)`.

4. A circle with radius r has **diameter** $d = 2 \times r$.

[9] `acos` is the C/C++ function name for mathematical function arccos.

5. A circle with diameter d has **circumference** (or **perimeter**) $c = \pi \times d$.
6. A circle with radius r has **area** $A = \pi \times r^2$

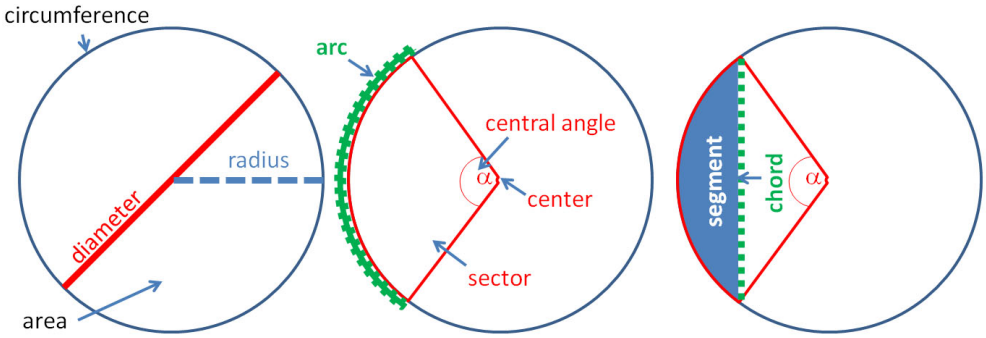


Figure 7.2: Circles

7. **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 7.2, right) in degrees[10], we can compute the length of the corresponding arc as $\frac{\alpha}{360.0} \times c$.
8. **Chord** of a circle is defined a line segment whose endpoints lie on the circle (diameter is the longest chord in a circle). A circle with radius r and a central angle α (in degrees) – see Figure 7.2, right – has the corresponding chord with length $sqrt(2r^2 \times (1 - cos(\alpha)))$. This can be derived from the **Law of Cosines** – see the explanation of this law in the discussion about Triangles below. 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.
9. **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 7.2, right – has the corresponding sector area $\frac{\alpha}{360.0} \times A$.
10. **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 7.2, 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.
11. 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 7.3). The code is shown below.

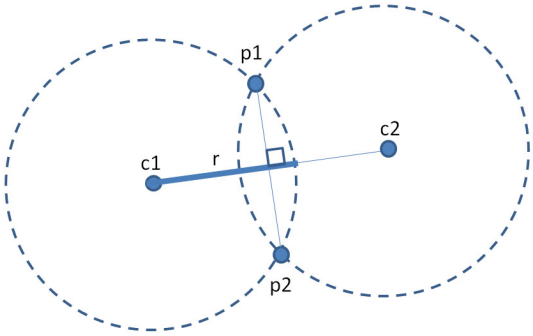


Figure 7.3: Circle Through 2 Points and Radius

Example codes: `ch7_02_circles.cpp`; `ch7_02_circles.java`

[10]Human usually works with degrees, but many mathematical functions in most programming languages work with radians. Check your programming language manual to verify this (at least C/C++/Java do). To help with conversion, just remember that one π radian equals to 180 degrees.

Exercise 7.2.3.1: Explain what is computed by the code below!

```
bool circle2PtsRad(point p1, point p2, double r, point *c) { // answer at *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
```

Programming Exercises related to Circles (only):

1. **UVa 10005 - Packing polygons *** (complete search; `circle2PointsRadius`)
2. **UVa 10012 - How Big Is It? *** (try all 8! permutations, Euclidean distance)
3. UVa 10136 - Chocolate Chip Cookies (`circle2PointsRadius`, 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 10301 - Rings and Glue (circle-circle intersection, backtracking)
8. UVa 10432 - Polygon Inside A Circle (area of n-sided regular polygon inside circle)
9. **UVa 10451 - Ancient Village ... *** (inner/outer circle of n-sided regular 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 *ellipse*, a generalization of the formula for a circle)
13. UVa 11515 - Cranes (circle-circle intersection, backtracking)

7.2.4 2D Objects: Triangles

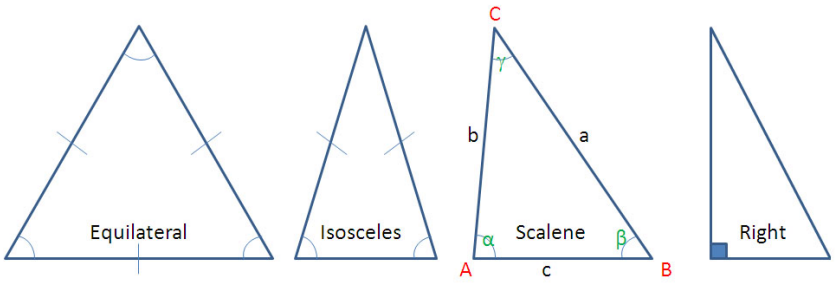


Figure 7.4: Triangles

1. **Triangle** (three angles) is a polygon (defined below) with three vertices and three edges. There are several types of triangles:
   a. **Equilateral** Triangle, all three edges have the same length and all inside (interior) angles are 60 degrees;
   b. **Isosceles** Triangle, two edges have the same length and two interior angles are the same.
   c. **Scalene** Triangle, all edges have different length;
   d. **Right** Triangle, *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**.

   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 above without experiencing overflow (assuming that we use 64-bit integers)? What should we do to avoid this issue?

5. A triangle with area A and semi-perimeter s has an **inscribed circle (incircle)** with radius $r = A/s$. The center of incircle is the meeting point between the triangle's *angle bisectors*.

6. A triangle with 3 sides: a, b, c and area A has an **circumscribed circle (circumcircle)** with radius $R = a \times b \times c/(4 \times A)$. The center of circumcircle is the meeting point between the triangle's *perpendicular bisectors*.

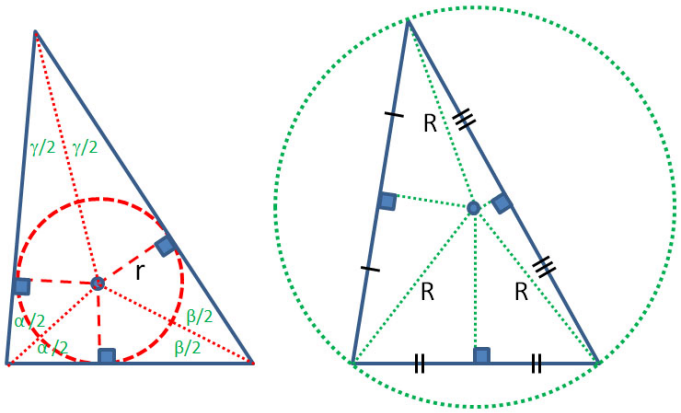


Figure 7.5: Incircle and Circumcircle of a Triangle

7. 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 do not make up 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 $(a + b > c)$.

8. 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 7.4. With the notations described there, we have: $c^2 = a^2 + b^2 - 2 \times a \times b \times cos(\gamma)$, or $\gamma = acos(\frac{a^2+b^2-c^2}{2 \times a \times b})$.

   Exercise 7.2.4.2: Rewrite this Cosine Formula for the other two angles α and β!

9. 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 7.4. With the notations described there, we have: $\frac{a}{\sin(\alpha)} = \frac{b}{\sin(\beta)} = \frac{c}{\sin(\gamma)}$.

10. In Trigonometry, the **Pythagorean Theorem** specializes the Law of Cosines. Pythagorean 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.

11. The **Pythagorean Triple** is a triple with three positive integers a, b, and c, such that $a^2 + b^2 = c^2$. Such a triple is commonly written as (a, b, c). 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. Pythagorean Triples describe the integer lengths of the three sides of a Right Triangle.

Example codes: `ch7_03_triangles.cpp`; `ch7_03_triangles.java`

Programming Exercises related to Triangles (plus Circles):

1. UVa 00143 - Orchard Trees (count int points in triangle; be careful with precision)
2. UVa 00190 - Circle Through Three Points (triangle's circumcircle)
3. UVa 00375 - Inscribed Circles and ... (triangle's incircle**s**!)
4. UVa 00438 - The Circumference of ... (triangle's circumcircle)
5. UVa 10195 - The Knights Of The Round ... (triangle's incircle, Heron's formula)
6. UVa 10210 - Romeo & Juliet (basic trigonometry)
7. UVa 10286 - The Trouble with a Pentagon (Law of Sines)
8. UVa 10347 - Medians (given 3 medians of a triangle, find its area)
9. UVa 10387 - Billiard (expanding surface, *trigonometry*)
10. UVa 10522 - Height to Area (derive the formula, uses Heron's formula)
11. **UVa 10577 - Bounding box *** (get center + radius of outer circle from 3 points[11])
12. UVa 10991 - Region (Heron's formula, Law of Cosines, area of sector)
13. **UVa 11152 - Colourful Flowers *** (triangle's (in/circum)circle; Heron's formula)
14. UVa 11479 - Is this the easiest problem? (property check)
15. UVa 11854 - Egypt (Pythagorean theorem/triple)
16. **UVa 11909 - Soya Milk *** (can use Law of Sines (or tangent); two possible cases!)
17. UVa 11936 - The Lazy Lumberjacks (check if 3 given sides can form a valid triangle)
18. LA 4413 - Triangle Hazard (KualaLumpur08, circles in triangles)
19. LA 4717 - In-circles Again (Phuket09, circles in triangles)

7.2.5 2D Objects: Quadrilaterals

1. **Quadrilateral** or **Quadrangle** is a polygon with with four edges (and four vertices). The term 'polygon' itself is described in more details below (Section 7.3). Figure 7.6 shows 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 with opposite sides parallel.

[11] After that, obtain all vertices, then get the min-x/max-x/min-y/max-y of the polygon.

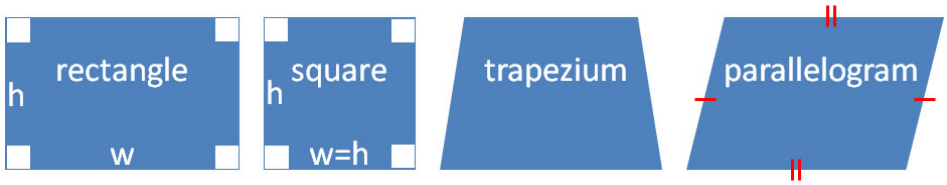


Figure 7.6: Quadrilaterals

Programming Exercises related to Quadrilaterals:

1. UVa 00155 - All Squares (recursive counting)
2. UVa 00201 - Square (counting square of various sizes; try all)
3. **UVa 00460 - Overlapping Rectangles *** (rectangle-rectangle intersection)
4. UVa 00476 - Points in Figures: Rectangles (similar to UVa 477 and 478)
5. UVa 00477 - Points in Figures: ... (similar to UVa 476 and 478)
6. UVa 10502 - Counting Rectangles (complete search)
7. UVa 10823 - Of Circles and Squares (complete search; check if point inside circles/squares)
8. UVa 10908 - Largest Square (implicit square)
9. **UVa 11207 - The Easiest Way *** (cutting rectangle into 4-equal-sized squares)
10. UVa 11345 - Rectangles (rectangle-rectangle intersection)
11. UVa 11455 - Behold My Quadrangle (property check)
12. UVa 11639 - Guard the Land (rectangle-rectangle intersection, use flag array)
13. **UVa 11834 - Elevator *** (packing two circles in a rectangle)

7.2.6 3D Objects: Spheres

1. **Sphere** is a perfectly round geometrical object in 3D space.
2. 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 7.7 left and middle).

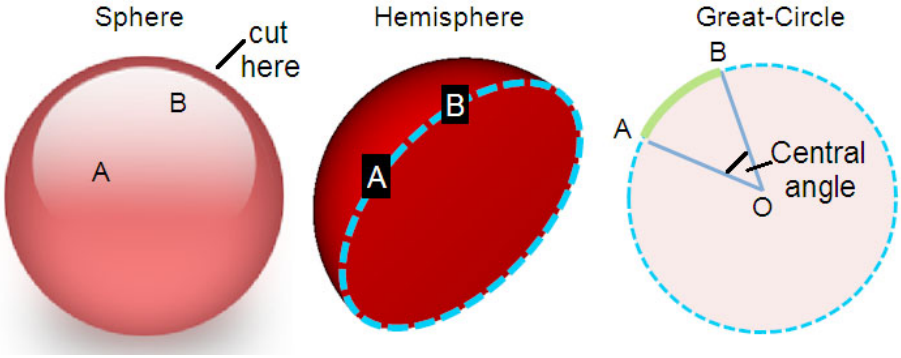


Figure 7.7: Left: Sphere, Middle: Hemisphere and Great-Circle, Right gcDistance (Arc A-B)

To find the Great-Circle Distance, we have to find the central angle AOB (see Figure 7.7, 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, the (latitude, longitude) pair. The following library code will help us 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;   // conversion from 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));
}
```

Example codes: ch7_04_UVa11817.cpp; ch7_04_UVa11817.java

Programming Exercises related to the Great-Circle Distance:

1. **UVa 00535 - Globetrotter *** (gcDistance)
2. **UVa 10075 - Airlines *** (gcDistance with APSP, see Section 4.5)
3. UVa 10316 - Airline Hub (gcDistance)
4. UVa 10897 - Travelling Distance (gcDistance)
5. **UVa 11817 - Tunnelling The Earth *** (gcDistance plus 3D Euclidean distance)

7.2.7 3D Objects: Others

Programming contest problem involving 3D objects is rare. But when it appears in a problem set, it can be one of the hardest. We provide an initial list of problems involving 3D objects below.

Programming Exercises related to Other 3D Objects:

1. **UVa 00737 - Gleaming the Cubes *** (cube and cube intersection)
2. **UVa 00815 - Flooded *** (volume, greedy, sort based on height, simulation)
3. **UVa 10297 - Beavergnaw *** (cones, cylinders, volumes)

Profile of Algorithm Inventors

Pythagoras of Samos (≈ 500 BC) was a Greek mathematician and philosopher. He was born on the island of Samos. He is best known for the Pythagorean theorem involving right triangle.

Euclid of Alexandria (≈ 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 (≈ 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.

7.3 Polygons 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 geometry problems as it allows the problem setter to present more realistic objects.

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 equals to the last vertex (some of the functions mentioned later require this arrangement).

```
struct point { double x, y;                                    // reproduced here
  point(double _x, double _y) { x = _x, y = _y; } };

// 6 points, entered in counter clockwise order, 0-based indexing
vector<point> P;                                    //7 P5--------------P4
P.push_back(point(1, 1));                           //6 |                 \
P.push_back(point(3, 3));                           //5 |                   \
P.push_back(point(9, 1));                           //4 |                    P3
P.push_back(point(12, 4));                          //3 |   P1___            /
P.push_back(point(9, 7));                           //2 | /       \ ___    /
P.push_back(point(1, 7));                           //1 P0              P2
P.push_back(P[0]);   // important: loop back        //0 1 2 3 4 5 6 7 8 9 101112
```

7.3.2 Perimeter of a Polygon

The perimeter of polygon with n vertices given in some order (clockwise or counter-clockwise) can be computed via this simple function below.

```
double dist(point p1, point p2) {        // get Euclidean distance of two points
  return hypot(p1.x - p2.x, p1.y - p2.y); }                  // as shown earlier

// returns the perimeter, which is the sum of Euclidian distances
// of consecutive line segments (polygon edges)
double perimeter(vector<point> P) {
  double result = 0.0;
  for (int i = 0; i < (int)P.size() - 1; i++)    // assume that the first vertex
    result += dist(P[i], P[(i + 1)]);             // is equal to the last vertex
  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 (clockwise or counter-clockwise) is:

```
         |   x0     y0     |
         |   x1     y1     |
         |   x2     y2     |
     1   |   x3     y3     |
A  = - * |   .      .      | = 1/2 * (x0y1 + x1y2 + x2y3 + ... + x(n-1)y0
     2   |   .      .      |         -x1y0 - x2y1 - x3y2 - ... - x0y(n-1))
         |   .      .      |
         |   x(n-1) y(n-1) |
         |   x0     y0     | <-- cycle back to the first vertex
```

This formula can be written into the library code below.

```
// returns the area, which is half the determinant
double area(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;              // assume that the first vertex
    y1 = P[i].y; y2 = P[(i + 1)].y;               // is equal to the last vertex
    result += (x1 * y2 - x2 * y1);
  }
  return fabs(result) / 2.0; }
```

Exercise 7.3.3.1: If the first vertex is not repeated as the last vertex, will the function `perimeter` and/or `area` presented as above work correctly?

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**.

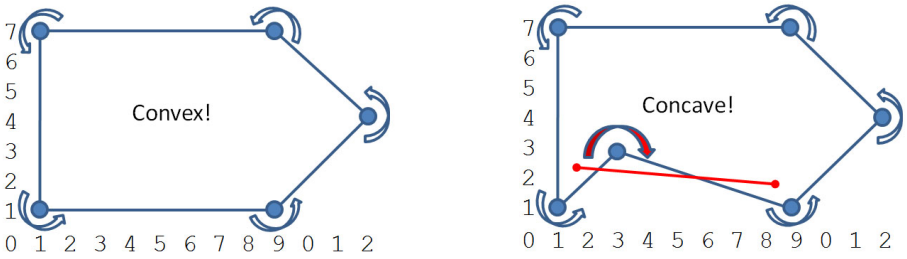


Figure 7.8: Left: Convex Polygon, Right: Concave Polygon

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).

```
// returns true if all three consecutive vertices of P form the same turns
bool isConvex(vector<point> P) {
  int sz = (int)P.size();
  if (sz < 3)         // boundary case, we treat a point or a line as not convex
    return false;
  bool isLeft = ccw(P[0], P[1], P[2]);              // remember one turn result
  for (int i = 1; i < (int)P.size(); i++)        // then compare with the others
    if (ccw(P[i], P[(i + 1) % sz], P[(i + 2) % sz]) != isLeft)
      return false;           // if different sign, then this polygon is concave
  return true; }                                        // this polygon is convex
```

7.3.5 Checking if a Point is Inside a Polygon

Another common test performed on polygon is to check if a point p is inside or outside the polygon. The following function allows such check for *either* convex or concave polygon. It works by computing the sum of angles between p and consecutive sides of the polygon, taking care of left turns (add the angle) and right turns (subtract the angle) respectively. If the final sum is 2π (360 degrees), then p is inside the polygon (see Figure 7.9).

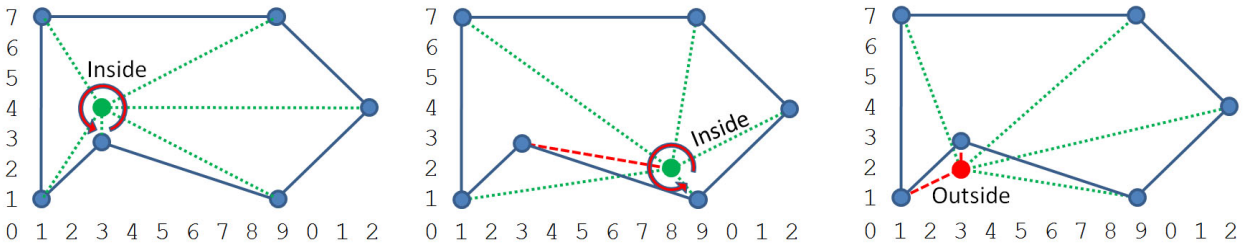


Figure 7.9: Left: inside, Middle: also inside, Right: outside

```
double angle(point a, point b, point c) {
  double ux = b.x - a.x, uy = b.y - a.y;
  double vx = c.x - a.x, vy = c.y - a.y;
  return acos((ux*vx + uy*vy) /
              sqrt((ux*ux + uy*uy) * (vx*vx + vy*vy))); }

// returns true if point p is in either convex/concave polygon P
bool inPolygon(point p, vector<point> P) {
  if ((int)P.size() == 0) return false;
  double sum = 0;
  for (int i = 0; i < (int)P.size() - 1; i++) {  // assume that the first vertex
    if (cross(p, P[i], P[i + 1]) < 0)             // is equal to the last vertex
      sum -= angle(p, P[i], P[i + 1]);                        // right turn/cw
    else sum += angle(p, P[i], P[i + 1]); }                   // left turn/ccw
  return (fabs(sum - 2*PI) < EPS || fabs(sum + 2*PI) < EPS); }
```

7.3.6 Cutting Polygon with a Straight Line

Another interesting thing that we can do with polygon is to cut it into two components with a straight line defined with two points a and b. See some programming exercises listed below that use this function.

The basic idea is to iterate through the vertices of the original polygon Q one by one. If a polygon vertex v and line ab form a left turn (v is on the left side of the straight line), 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, 'intersection 1'). We then skip the next few vertices of P as those vertices will be on the right side of line ab. Sooner or later, we will revisit another edge that intersect with line ab again (see Figure 7.10, left, 'intersection 2'). 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 (see Figure 7.10).

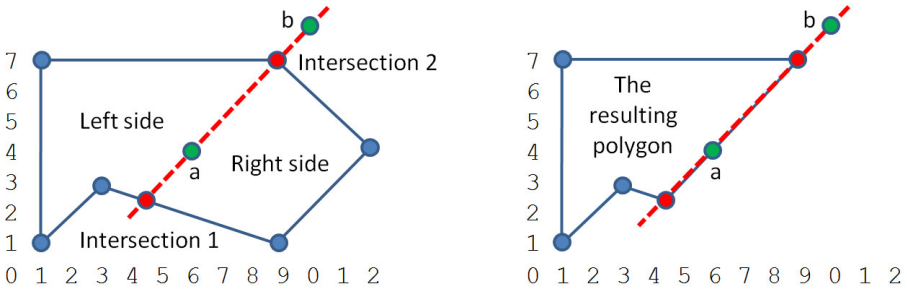


Figure 7.10: Left: Before Cut, Right: After Cut

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?

```
// 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, vector<point> Q) {
  vector<point> P;
  for (int i = 0; i < (int)Q.size(); i++) {
    double left1 = cross(a, b, Q[i]), left2 = 0.0;
    if (i != (int)Q.size() - 1) left2 = cross(a, b, Q[i + 1]);
    if (left1 > -EPS) P.push_back(Q[i]);
    if (left1 * left2 < -EPS)
      P.push_back(lineIntersectSeg(Q[i], Q[i + 1], a, b));
  }
  if (P.empty()) return P;
  if (fabs(P.back().x - P.front().x) > EPS ||
      fabs(P.back().y - P.front().y) > EPS)
    P.push_back(P.front());
  return P; }
```

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 smallest 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.

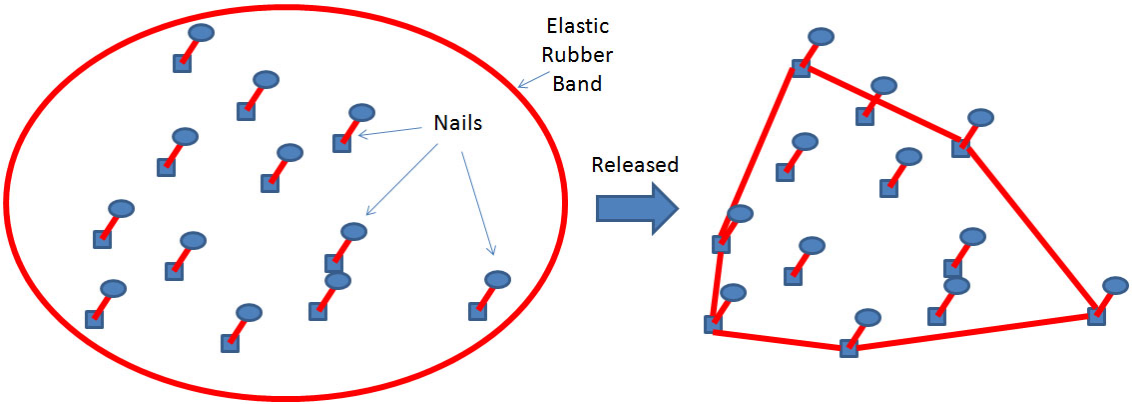


Figure 7.11: Rubber Band Analogy for Finding Convex Hull

As every vertex in $CH(P)$ is a vertex in the original polygon P, the algorithm for finding convex hull is essentially an algorithm to decide which vertices 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 (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)!

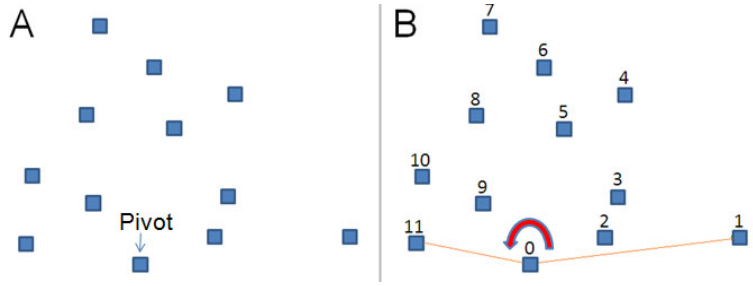


Figure 7.12: Sorting Set of Points by Their Angles w.r.t a Pivot (Point 0)

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.

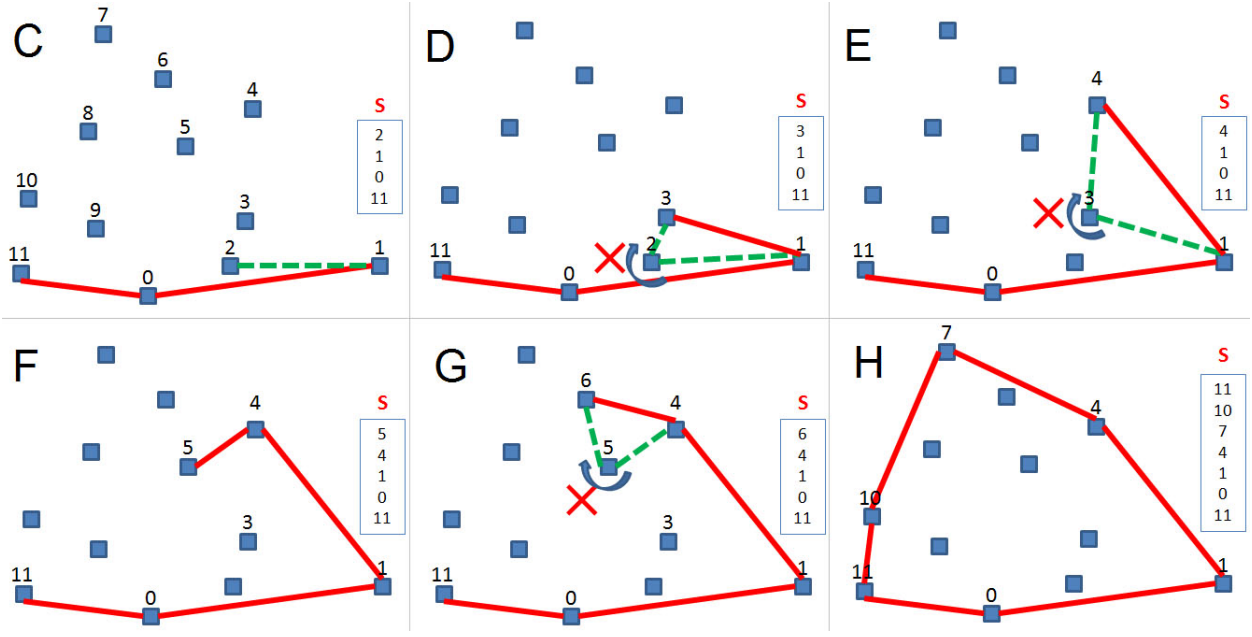


Figure 7.13: The Main Part of Graham's Scan algorithm

The implementation of Graham's Scan algorithm is shown below.

```
point pivot(0, 0);
bool angleCmp(point a, point b) {                        // angle-sorting function
  if (collinear(pivot, a, b))
    return dist(pivot, a) < dist(pivot, b);     // determine 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 the two angles

vector<point> CH(vector<point> P) {
  int i, N = (int)P.size();
  if (N <= 3) 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;
  // swap selected vertex with P[0]
  point temp = P[0]; P[0] = P[P0]; P[P0] = temp;

  // 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);     // notice that we does not sort P[0]

  // third, the ccw tests
  point prev(0, 0), now(0, 0);
  stack<point> S; S.push(P[N - 1]); S.push(P[0]);  // initial content of stack S
  i = 1;                                             // then, we check the rest
  while (i < N) {                // note: N must be >= 3 for this method to work
    now = S.top();
    S.pop(); prev = S.top(); S.push(now);                    // get 2nd from top
    if (ccw(prev, now, P[i])) S.push(P[i++]);               // left turn, accept
    else S.pop(); // otherwise, pop the top of stack S until we have a left turn
  }

  vector<point> ConvexHull;                           // from stack back to vector
  while (!S.empty()) { ConvexHull.push_back(S.top()); S.pop(); }
  return ConvexHull; }                                          // return the result
```

Example codes: ch7_05_polygon.cpp; ch7_05_polygon.java

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 $O(n \log n)$.

Exercise 7.3.7.1: Suppose we have 5 points: {(0, 0), (1, 0), (2, 0), (2, 2), (0, 2)}. The convex hull of these 5 points are actually these 5 points themselves. 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 library function: `atan2`. This function is used to compare the two angles but what is actually returned by `atan2`? Investigate!

Exercise 7.3.7.3: 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 eschews costly comparisons between angles [5]. Investigate this algorithm and implement it!

Below, we provide a list of programming exercises related to polygon. Without pre-written library codes discussed in this section, many of these problems look 'hard'. With the library codes, 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 00218 - Moth Eradication (find `CH`, `perimeter` of polygon)
3. UVa 00361 - Cops and Robbers (check if a point is inside `CH` of Cop/Robber[12])
4. UVa 00478 - Points in Figures: ... (test if point `inPolygon`/triangle[13])
5. UVa 00634 - Polygon (test if point `inPolygon`, the polygon can be convex or concave)
6. UVa 00681 - Convex Hull Finding (pure `CH` problem)
7. **UVa 00811 - The Fortified Forest *** (`CH`, `perimeter` of polygon, complete search)
8. UVa 00858 - Berry Picking (ver line-polygon intersect; sort; alternating segments)
9. UVa 10060 - A Hole to Catch a Man (`area` of polygon)
10. UVa 10065 - Useless Tile Packers (find `CH`, `area` of polygon)
11. UVa 10078 - Art Gallery (test if polygon is concave (i.e. not `isConvex`))
12. UVa 10088 - Trees on My Island (Georg A. Pick's Theorem[14]: $A = i + \frac{b}{2} - 1$)
13. UVa 10112 - Myacm Triangles (test if point `inPolygon`/triangle, similar to UVa 478)
14. UVa 10406 - Cutting tabletops (vector, `rotate`, `translate`, then `cutPolygon`)
15. **UVa 10652 - Board Wrapping *** (`rotate`, `translate`, `CH`, `area`)
16. UVa 11096 - Nails (very classic `CH` problem, start from here)
17. **UVa 11265 - The Sultan's Problem *** (`cutPolygon`, test if point `inPolygon`, get `area`)
18. UVa 11447 - Reservoir Logs (`area` of polygon)
19. UVa 11473 - Campus Roads (`perimeter` of polygon)
20. UVa 11626 - Convex Hull (find `CH`, be careful with collinear points)
21. LA 3169 - Boundary Points (Manila06, convex hull `CH`)

Profile of Algorithm Inventor

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. Since then, there are many other algorithm variants and improvements for finding convex hull.

[12]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.

[13]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.

[14]More details about this Pick's theorem: I is the number of integer points in the polygon, A is the area of the polygon, and B is the number of integer points on the boundary.

7.4 Divide and Conquer Revisited

Several computational geometry problems turn out to be solvable with Divide and Conquer paradigm that has been elaborated earlier in Section 3.3. Below is one example of such application.

Bisection Method for Geometry Problem

We use UVa 11646 - Athletics Track for illustration. The abridged problem description is as follows: Examine a rectangular soccer field with an athletics track as seen in Figure 7.14, 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?

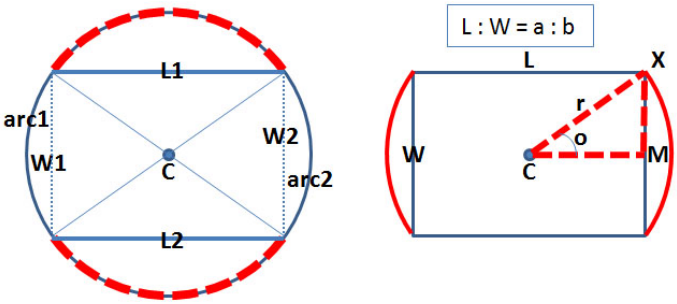


Figure 7.14: Athletics Track (from UVa 11646)

It is quite hard to find the solution with pen and paper, but with the help of a computer and bisection method (binary search), 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 7.14, 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; // angle of the arc is twice 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);
```

Programming Exercises related to Divide and Conquer:

1. **UVa 00152 - Tree's a Crowd *** (sort 3D, partition the space)
2. **UVa 10245 - The Closest Pair Problem *** (classic, as the problem name implies)
3. UVa 10566 - Crossed Ladders (bisection method)
4. UVa 10668 - Expanding Rods (bisection method)
5. UVa 11378 - Bey Battle (also a closest pair problem)
6. **UVa 11646 - Athletics Track *** (bisection method, the circle is at the center of track)

7.5 Chapter Notes

Some materials in this chapter are derived from the materials courtesy of **Dr Cheng Holun, Alan** from School of Computing, National University of Singapore. Some other library functions are customized from **Igor Naverniouk** code library: `http://shygypsy.com/tools/`.

Compared to the first edition of this book, this chapter has grown almost twice the size. Similar case as with Chapter 5 and 6. 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: [30, 5, 3]. But not just the theory, he must also train himself to code *robust* geometry solution that is 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.

There are $\approx$ **99 UVa (+ 4 others) programming exercises** discussed in this chapter.
(Only 96 in the first edition, a 7% increase).

There are **22 pages** in this chapter.
(Only 13 in the first edition, a 69% increase).

Chapter 8

More Advanced Topics

Genius is one percent inspiration, ninety-nine percent perspiration.
— **Thomas Alva Edison**

8.1 Overview and Motivation

The purpose of having this chapter is twofold. First, this chapter allows us to take out the harder material from the earlier chapters. Section 8.2 contains discussions of many problems which can be confusing for new competitive programmers if they are listed in the earlier chapters. This is because the problems listed in Section 8.2 require more than one data structures and/or algorithms. It is more appropriate to discuss them in this chapter, after all the various data structures and algorithms have been discussed. Therefore, it is a good idea to master Chapter 1-7 first before reading this chapter.

The second purpose is for easier extension in the future editions of this book. The style of this chapter is currently as follow: Three major themes in three sections: Problem Decomposition (Section 8.2), More Advanced Search Techniques (Section 8.3), and More Advanced Dynamic Programming Techniques (Section 8.4). Each section contains independent write ups of some harder problems. This style makes this chapter easy to extend in the future.

8.2 Problem Decomposition

While there are only 'a few' basic data structures and algorithms tested in contest problems (most of them are covered in this book), harder problems may require a *combination* of two (or more) data structures and/or algorithms. For such problems, try to decompose parts of the problems so that you can solve the different parts independently. To be able to decompose a problem into its components, we must first be familiar with the individual components. We illustrate this decomposition technique using several examples.

8.2.1 Two Components: Binary Search the Answer and Other

In Section 3.3.1, we have seen binary search the answer on a simulation problem. Actually, this technique can be combined with other algorithms.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, LA 2949, 5000,
- MCBM (discussed in Section 4.7.4), e.g. UVa 10804, 11262,
- SSSP algorithm (discussed in Section 4.4), e.g. UVa 10816, IOI 2009 (Mecho),
- Max Flow (discussed in Section 4.6), e.g. UVa 10983.

The rest of this write up is a detailed solution for UVa 714 - Copying Books. This problem can be solved with binary search the answer technique plus Greedy algorithm.

Abridged problem description: 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 $0 = b_0 < b_1 < b_2, \ldots < b_{k-1} \leq b_k = m$ such that i-th scriber 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?".

There exists a Dynamic Programming solution for this problem (as $m \leq 500$), but this problem can already 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!

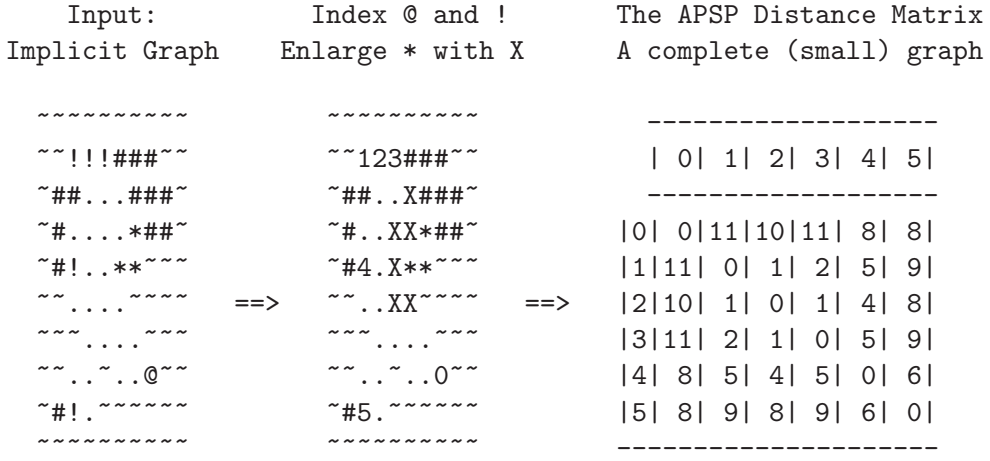
If we guess *answer* = 2000, then we can greedily assign the jobs as follows: {100, 200, 300, 400, 500} for scribe 1, {600, 700} for scribe 2, and {800, 900} for scribe 3. All books are copied and we still have some slacks, i.e. scribe 1, 2, and 3 still have {500, 700, 300} unused potential. Therefore 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 $hi = p_1 + p_2 + \ldots + p_m$ (all pages from all books).

For those who are curious, the optimal value of *answer* for the test case shown in this example is *answer* = 1700.

8.2.2 Two Components: SSSP and DP

In this write up, we want to highlight a problem where SSSP is one of the component and DP is the other component. The SSSP is usually used to transform the input (usually an implicit graph/grid) into another (smaller) graph. Then we do Dynamic Programming on the second (usually smaller) graph. We use UVa 10937 - Blackbeard the Pirate to illustrate this combination.



The given input for this problem is shown on the left. 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 them in any of 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.

This is clearly a TSP problem (see Section 3.5.3), but before we can do that, we have to first transform the input into a distance matrix.

In this problem, we are only interested in '@' and the '!'s. We give index 0 to '@' and give positive indices to the other '!'s. We enlarge the reach of '*' with 'X'. Then we run BFS on this unweighted implicit graph starting from each '@' or '!', by only stepping on cells labeled with '.' (land cells). This gives us the All-Pairs Shortest Paths (APSP) distance matrix as shown 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 TSP tour is: 0-5-4-1-2-3-0 with cost = 8+6+5+1+1+11 = 32.

8.2.3 Two Components: Involving Graph

Some modern problems involving *directed* graph deal with Strongly Connected Components (SCCs). 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 (another name: super vertices). Note that if the SCCs of a directed graph are contracted, the resulting graph of super vertices is a DAG (see Figure 4.8 for an example). If you recall our discussion in Section 4.7.1, DAG is very suitable for DP techniques.

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 represent the number of original vertices that are contracted into that super vertex.

8.2.4 Two Components: Involving Mathematics

In Section 2.3.1, we mention that for some problem, the underlying graph does not need to be stored in any graph specific data structures. 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.

8.2.5 Three Components: 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 parts.

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 $N!$.

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.".

8.2.6 Three Components: Complete Search, Binary Search, Greedy

In this write up, we discuss a recent top-level programming problems 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. Then, your task is to:

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 8.1 for illustration:
line = the safe landing time window of a plane
star = the plane's optimal landing schedule.

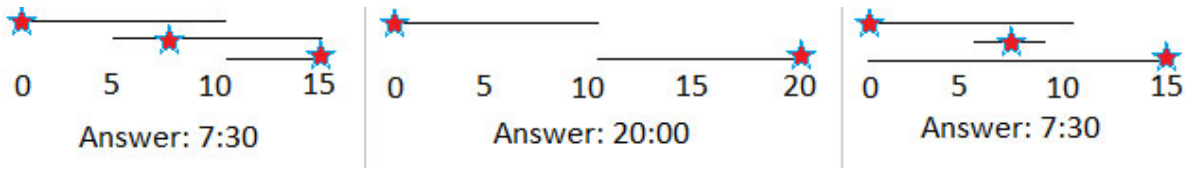


Figure 8.1: Illustration for ACM ICPC WF2009 - A - A Careful Approach

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 implementing 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 error $\epsilon <$ 1e-3 is enough. For more details, please study our Accepted source code shown below.

```
/* World Finals Stockholm 2009, A - A Careful Approach, LA 4445 (Accepted) */

#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, first aircraft lands immediately
  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; } // returning +ve value will force binary search to reduce L
  // returning -ve value will 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 can 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
      // This example code uses 'double' data type. This is actually not a good
      // practice. Some other programmers avoid the test below and simply use
      // 'loop 100 times (precise enough)'
      while (fabs(lo - hi) >= 1e-3) {     // binary search L, ERROR = 1e-3 is OK
        L = (lo + hi) / 2.0;  // we want the answer to be 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

    // another 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;
```

Example codes: ch8_01_LA4445.cpp; ch8_01_LA4445.java

Exercise 8.2.1: 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 10804 - Gopher Strategy (similar to UVa 11262)
  3. **UVa 10816 - Travel in Desert *** (binary search the answer (temp) + Dijkstra's)
  4. **UVa 10983 - Buy one, get the ... *** (binary search the answer + max flow)

5. **UVa 11262 - Weird Fence *** (binary search the answer + bipartite matching[1])
6. UVa 11516 - WiFi (binary search the answer + greedy)
7. LA 2949 - Elevator Stopping Plan (Guangzhou03, binary search the answer + greedy)
8. LA 3795 - Against Mammoths (Tehran06)
9. LA 5000 - Underwater Snipers (KualaLumpur10, binary search + greedy)
10. IOI 2009 - Mecho (binary search the answer + BFS)

- Two Components - Involving DP 1D Range Sum
  1. UVa 00967 - Circular (similar to UVa 897; speed up with DP 1D range sum)
  2. UVa 10533 - Digit Primes (sieve; check if a prime is a digit prime; DP 1D range sum)
  3. **UVa 10871 - Primed Subsequence *** (need 1D Range Sum Query)
  4. **UVa 10891 - Game of Sum *** (Double DP[2])
  5. **UVa 11408 - Count DePrimes *** (need 1D Range Sum Query)
- Two Components - SSSP and DP
  1. UVa 10917 - A Walk Through the Forest (counting paths in DAG[3]; DP)
  2. **UVa 10937 - Blackbeard the Pirate *** (BFS → TSP, then DP or backtracking)
  3. UVa 10944 - Nuts for nuts.. (BFS → TSP, then use DP, $n \leq 16$)
  4. **UVa 11405 - Can U Win? *** (BFS from 'k' & each 'P' – max 9 items; DP-TSP)
  5. **UVa 11813 - Shopping *** (Dijsktra's → TSP, then use DP, $n \leq 10$)
- Two Components - Involving Graph
  1. UVa 10307 - Killing Aliens in Borg Maze (build SSSP graph with BFS, MST)
  2. UVa 11267 - The 'Hire-a-Coder' ... (bipartite check, MST accept -ve weight)
  3. **UVa 11324 - The Largest Clique *** (longest paths on DAG[4]; toposort; DP)
  4. **UVa 11635 - Hotel Booking *** (Dijkstra's + BFS)
  5. **UVa 11721 - Instant View ... *** (find nodes that can reach SCCs with neg cycle)
  6. LA 3290 - Invite Your Friends (Dhaka05, BFS + Dijkstra's)
  7. LA 3294 - The ... Bamboo Eater (Dhaka05, longest path, DAG, 2D Segment Tree)
  8. LA 4272 - Polynomial-time Red... (Hefei08, SCC)
  9. LA 4407 - Gun Fight (KualaLumpur08, geometry, MCBM)
  10. LA 4607 - Robot Challenge (SoutheastUSA09, geometry, SSSP on DAG → DP)
  11. LA 4846 - Mines (Daejeon10, geometry, SCC, see UVa 11504 & 11770)
- Two Components - Involving Mathematics
  1. **UVa 10637 - Coprimes *** (involving prime numbers and gcd)
  2. **UVa 10717 - Mint *** (complete search + GCD/LCM, see Section 5.5.2)
  3. **UVa 11428 - Cubes *** (complete search + binary search)
  4. UVa 11730 - Number Transformation (needs prime factoring, see Section 5.5.1)
  5. LA 4203 - Puzzles of Triangles
- Three Components
  1. **UVa 10856 - Recover Factorial *** (discussed in this section)
  2. **UVa 11610 - Reverse Prime *** (see footnote for the components[5])
  3. LA 4445 - A Careful Approach (World Finals Stockholm09, discussed in this chapter)

---

[1]We can use the alternating path algorithm to compute the MCBM (see Section 4.7.4).

[2]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.

[3]But first, you have to build the DAG by running Dijkstra's algorithm from 'home'.

[4]But first, you have to transform the graph into DAG of its SCCs.

[5]First, reverse primes less than 10^6, append zero(es) if necessary. Use Fenwick Tree and binary search.

8.3 More Advanced Search Techniques

In this section, we discuss some more advanced search techniques that may be applicable to some harder problems.

8.3.1 Informed Search: A*

Figure 8.2: 15 Puzzle

In Chapter 3 and 4, we have seen several graph traversal/state search algorithms: recursive backtracking/modified BFS. These algorithms are 'uninformed', i.e. all possible states reachable from the current state are equally good. For some problems, we do have more information and we can use A* search that employs heuristic to 'guide' the search direction.

We will illustrate this A* search using a well-known 15-puzzle problem. There are 15 sliding tiles in the puzzle, each with a number from 1 to 15 on it. These 15 tiles are packed into a 4 by 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 are ordered as in Figure 8.2, the 'goal' state.

This seemingly small puzzle is a headache for 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's sake, we assign the blank tile value 0, so the goal state will be {1, 2, 3, ..., 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 that can help guiding the search algorithm, which is the sum of Manhattan distances between each 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 below:

```
  The current/          The state s' if we    The state s* if we    The state s+ if we
starting state s:        slide 0 upwards:      slide 0 leftwards:    slide 0 downwards:
   1  2  3  4              1  2  3  4            1  2  3  4            1  2  3  4
   5  6  7  8              5  6  7  0            5  6  7  8            5  6  7  8
   9 10 11  0              9 10 11  8            9 10  0 11            9 10 11 12
  13 14 15 12             13 14 15 12           13 14 15 12           13 14 15  0

 g(s)=0, h(s)=1       g(s')=1, h(s')=2      g(s*)=1, h(s*)=2      g(s+)=1, h(s+)=0
                        g(s')+h(s')=3         g(s*)+h(s*)=3         g(s+)+h(s+)=1
```

The cost of the starting state is $g(s) = 0$, no move yet. There are three reachable states from this state (as tile '0' is at non-corner side), $g(s') = g(s*) = g(s+) = 1$:

1. The heuristic value if we slide tile 0 upwards is $h(s') = 2$ as tile 8 and tile 12 are both off by 1. This causes $g(s') + h(s') = 0 + 2 = 2$.
2. The heuristic value if we slide tile 0 leftwards is $h(s*) = 2$ as tile 11 and tile 12 are both off by 1. This causes $g(s*) + h(s*) = 0 + 2 = 2$.
3. But if we slide tile 0 downwards, we have $h(s+) = 0$. This causes $g(s+) + h(s+) = 0 + 0 = 0$.

If we visit the states in ascending order of $g(s) + h(s)$ values, we will explore the states with smaller expected cost first (i.e. state '$s+$' in this example). This is the essence of the A* search algorithm. We usually implement this state ordering with the help of a priority queue, as with Dijkstra's algorithm. Note that if $h(s)$ is set to 0 for all states, A* *degenerates* to a simple BFS 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* is finding such heuristic.

8.3.2 Depth Limited Search

In Section 3.2.1, we have seen recursive backtracking algorithm. Backtracking is similar to Depth First Search (DFS) but backtracking un-flags the visited vertices/states when it backtracks. This allows backtracking to explore the entire search space systematically. The main problem with pure backtracking is that: It may be trapped in a very deep path that will not lead to the solution, 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 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 backtracking. 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 backtracking.

8.3.3 Iterative Deepening Search

As said above, 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 BFS ability to visit neighboring states layer by layer (see Table 4.1 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. This is because $O(b^0 + (b^0 + b^1) + (b^0 + b^1 + b^2) + ... + (b^0 + b^1 + b^2 + ... + b^d)) \leq O(c \times b^d) = O(b^d)$.

8.3.4 Iterative Deepening A* (IDA*)

The problem with BFS/A* that uses queue/priority queue, respectively, is that the memory requirement can be very huge when the goal state is far from the initial state. In fact, UVa 10181 is not solvable with pure A* that uses priority queue to order the next states to be visited.

To solve UVa 10181, we need to use IDA* (Iterative Deepening A*) which is essentially IDS with modified DLS. IDA* calls modified DLS to try the all neighboring states in fixed order (i.e. slide tile 0 rightwards, then upwards, then leftwards, then finally downwards – in that order). 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.

Example codes (explore for details): `ch8_02_UVa10181.cpp`; `ch8_02_UVa10181.java`

Programming Exercises solvable with More Advanced Search Techniques:

1. UVa 00652 - Eight (classical sliding block 8-puzzle problem, IDA*)
2. UVa 10073 - Constrained Exchange Sort (IDA*)
3. **UVa 10181 - 15-Puzzle Problem *** (similar as UVa 652, but this one is larger, IDA*)
4. **UVa 11163 - Jaguar King *** (IDA*)
5. **UVa 11212 - Editing a Book *** (A*)
6. LA 3681 - Route Planning (Kaohsiung06, solvable[6] with IDA*)

[6]Test data likely only contains up to 15 stops which already include the starting and the last stop on the route.

8.4 More Advanced Dynamic Programming Techniques

In Section 3.5, 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 that section. Here, we present some of them.

8.4.1 Emerging Technique: DP + bitmask

Some of the modern DP[7] problems require a (small) set of Boolean to represent their state. This is where bitmask technique (see Section 2.2.1) can be useful. As a bitmask is actually just an integer, this technique is suitable for DP as the integer (that represents the bitmask) can be used as index of the DP table. We have seen this technique once when we discuss DP TSP (see Section 3.5.2). In this write up and the next one, we illustrate two other usages of this emerging technique.

Forming Quiz Teams (UVa 10911)

For the abridged problem statement and the full 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 Small General Graph". In general, this problem is hard and the solution is Edmond's Blossom algorithm [6] which is not easy to code. However, if the input size is small, up to $M \leq 20$, then DP + bitmask technique can be used.

The DP + 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 these simple bit operations, i.e. `bitmask | (1 << 0) | (1 << 2)`, thus the state becomes `bitmask=000101`. Notice that index starts from 0 and counted from the right. Then, if from this state, item 1 and item 5 are matched, the state becomes `bitmask=100111`. The perfect matching is obtained when the state is all '1's, in this case: `bitmask=111111`. This is similar with the DP TSP solution outlined in Section 3.5.2.

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. As we want a perfect matching, then for a currently 'off' bit i, we must find the best other 'off' bit j from `[`i`+1..`M`-1]` using one $O(M)$ loop. This check is again done using bit operation, i.e. `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 + bitmask approach is feasible. For more details, please study the code shown in Section 1.2.

Example codes: `ch8_03_UVa10911.cpp`; `ch8_03_UVa10911.java`

In this write up, we have shown that DP + 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 *lightweight dynamic set of Boolean* of up to ≈ 20 items. The programming exercises in this section contain more examples when bitmask is used as one of the parameter of the DP states.

Exercise 8.4.1.1: The code for UVa 10911 in Section 1.2 has a comment saying "This 'break' is necessary. Do you understand why?". Answer it!

8.4.2 Chinese Postman/Route Inspection Problem

The Chinese Postman[8]/Route Inspection Problem is the problem of finding 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 (covers all the edges in the Eulerian graph) is the optimal solution for this problem. This is the easy case. But when the

[7] Actually also applicable to other problems that use small set of Boolean.

[8] The name is because it is first studied by the Chinese mathematician Mei-Ku Kuan in 1962.

graph is non Eulerian, e.g. see the graph in Figure 8.3 (left), then this Chinese Postman Problem is harder.

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. This is an observation 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 according to the weight of the original edge (if it exists), or the shortest path weight of a path from i to j, e.g. edge `2-5` has weight $2 + 1 = 3$ from path `2-4-5`.

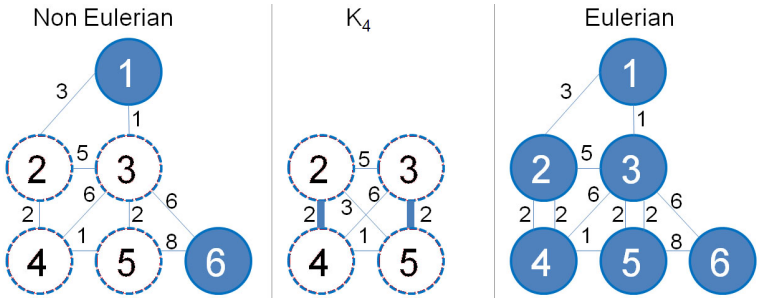


Figure 8.3: An Example of Chinese Postman Problem

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 8.3 (middle) is to take edge `2-4` (with weight 2) and edge `3-5` (also with weight 2).

After doubling edge `2-4` and edge `3-5`, we are now back to the easy case of the Chinese Postman Problem. In Figure 8.3 (right), we have an Eulerian graph. The Chinese Postman problem tour is simple in this Eulerian graph. One such tour is: `1->2->4->5->3->2->4->3->5->6->3->1` with total weight of `38` (the sum of all edge weight in the modified Eulerian graph).

The hardest part of solving the Chinese Postman Problem is in finding the minimum weight perfect matching on K_n. If n is small, this part can be solved with DP + bitmask technique shown in Section 8.4.1 above. Otherwise, we have to use Edmonds' Blossom algorithm [6].

8.4.3 Compilation of Common DP States

After solving lots of DP problems, contestants will develop a sense of which parameters are commonly selected to represent distinct states of a DP problem. Some of them are as follows:

1. Current position in an ordered array
   Original problem: $[x_1, x_2, \ldots, x_n]$,
   e.g. a list/sequence of numbers (integer/double), a string (character array)
   Sub problems: Break the original problem into:

   - Sub problem and suffix: $[x_1, x_2, \ldots, x_{n-1}] + x_n$
   - Prefix and sub problem: $x_1 + [x_2, x_3, \ldots, x_n]$
   - Two sub problems: $[x_1, x_2, \ldots, x_i] + [x_{i+1}, x_{i+2}, \ldots, x_n]$

   Example: LIS, 1D Max Sum, Matrix Chain Multiplication (MCM), etc (see Section 3.5.2)

2. Current positions in two ordered arrays (similar as above, but on two arrays)
   Original problem: $[x_1, x_2, \ldots, x_n]$ and $[y_1, y_2, \ldots, y_n]$,
   e.g. two lists/sequences/strings
   Sub problems: Break the original problem into:

- Sub problem and suffix:
  $[x_1, x_2, \ldots, x_{n-1}] + x_n$ and
  $[y_1, y_2, \ldots, y_{n-1}] + y_n$
- Prefix and sub problem:
  $x_1 + [x_2, x_3, \ldots, x_n]$ and
  $y_1 + [y_2, y_3, \ldots, y_n]$
- Two sub problems:
  $[x_1, x_2, \ldots, x_i] + [x_{i+1}, x_{i+2}, \ldots, x_n]$ and
  $[y_1, y_2, \ldots, y_j] + [x_{j+1}, x_{j+2}, \ldots, x_n]$

Example: String Alignment/Edit Distance, LCS, etc (see Section 6.5);
Note: Can also be applied to a 2D matrix, e.g. 2D Max Sum, etc

3. Vertices (Position) in Implicit/Explicit DAG
   Original problem: At certain vertex
   Sub problem: The vertices adjacent to the current vertices
   Example: Shortest/Longest Paths in DAG, Counting Paths on DAG, etc (see Section 4.7.1)
4. Knapsack-Style Parameter
   Original problem: Some given positive threshold value (or from zero)
   Sub problems: Decrease (or increase) current value until zero (or until reaching the threshold)
   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 8.4.6 if the values can go negative.
5. Subset (usually using bitmask technique)
   Original problem: Set S
   Subproblems: Break S into S_1 and S_2 where $S_1 \in S, S_2 \in S, S = S_1 \bigcup S_2$
   Example: DP-TSP (see Section 3.5.2), DP + bitmask (see Section 8.4.1), etc

Note that the harder DP problems usually combine two, three, or more parameters to represent distinct states. Try to solve more DP problems listed in this section to build your DP skills.

8.4.4 MLE/TLE? Use Better State Representation!

Do you receive Memory Limit Exceeded (MLE) verdict for your 'correct' DP solution (on small test cases)? Or if not MLE, you receive Time Limit Exceeded/TLE verdict instead? If that happens, one possible strategy is to use a better DP state representation in order to reduce the table size (and subsequently speed up the overall time complexity). We illustrate this using an example:

ACORN (ACM ICPC Singapore 2007, LA 4106)

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 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 in Figure 8.4, the best descent path has a total of 8 acorns (dotted line).

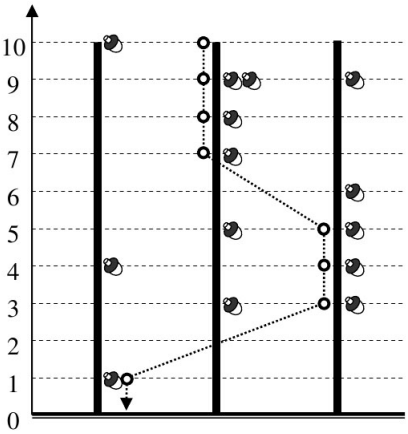


Figure 8.4: The Descent Path

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)); // best 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
```

Example codes: `ch8_04_LA4106.cpp`; `ch8_04_LA4106.java`

When the size of naïve DP states are too large that causes the overall DP time complexity not-doable, think of a more efficient (but usually not obvious) ways 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 setter must have known a trick.

8.4.5 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 a recent ACM ICPC world finals problem to illustrate this technique.

ACM ICPC World Finals 2010 - Problem J - Sharing Chocolate, 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 8.5 (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 8.5 (right).

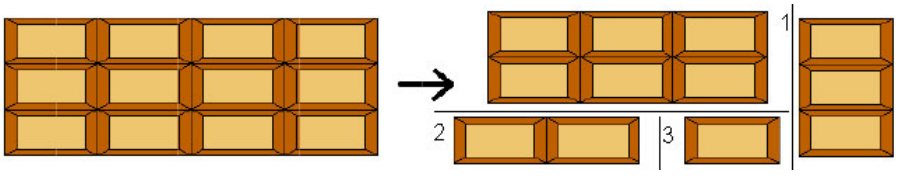


Figure 8.5: Illustration for ACM ICPC WF2010 - J - Sharing Chocolate

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 consider; 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 the $(w, bitmask)$ formulation. With this formulation, we can

'recover' the required value h via `sum(`*bitmask*`) / w`, where `sum(`*bitmask*`)` is the sum of the piece size 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.

Example codes: `ch8_05_LA4794.cpp`; `ch8_05_LA4794.java`

Exercise 8.4.5.1: 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 recovered from other parameters.

8.4.6 Your Parameter Values Go Negative? Use Offset Technique

In some rare cases, the possible range of a parameter used in DP states can go negative. This causes a problem for DP solution as the indices of a DP table must be non negative. Fortunately, this 'problem' can be dealt by using offset technique to make all indices non negative again. We illustrate this technique with another non trivial DP problem: Free Parentheses.

Free Parentheses (ACM ICPC Jakarta 2008, 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 it will reverse the meaning of subsequent '+' and '-' operators;
2. You can only put X close brackets if you already use X open brackets – we need to store this information to process the subproblems correctly;
3. The maximum value is 100 + 100 + ... 100 (30 times) = 3000 and the minimum value is 100 - 100 - ... - 100 (29 times) = -2800 – this information needs 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:
1. '`idx`' – the current position being processed, we need to know where we are now.
2. '`open`' – number of open brackets, we need this information in order to produce valid expression[9].

[9] 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))).

But these two parameters are not enough to uniquely identify sub-problems 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.

We can represent all possible states of this problem with `bool visited[idx][open][val]`, a 3-D array. The purpose of this memo table `visited` is to flag if certain state has been visited or not. As '`val`' ranges from -2800 to 3000 (5801 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 – fast enough. The code is shown below:

```
// 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)

void rec(int idx, int open, int val) {
  if (visited[idx][open][val+3000])        // this state has been reached before
    return; // notice 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
}

// The solution is all the values in array 'used' that are flagged as true.
// But remember to offset those values by -3000 first.
```

The DP formulation for this problem is not trivial. Try to find a state representation that can uniquely identify sub-problems. Make observations and consider attacking the problem from there.

Programming Exercises related to More Advanced DP:

- DP + bitmask
  1. UVa 10296 - Jogging Trails (Chinese Postman Problem, discussed in this section)
  2. UVa 10364 - Square (bitmask technique can be used)
  3. UVa 10651 - Pebble Solitaire (the problem size is small, still doable with backtracking)
  4. UVa 10817 - Headmaster's Headache (s: (id, bitmask of subjects), space: $100 \times 2^{2*8}$)
  5. **UVa 10911 - Forming Quiz Teams *** (elaborated in this section)
  6. UVa 11218 - KTV (still solvable with complete search)
  7. **UVa 11391 - Blobs in the Board *** (DP + bitmask on 2D grid)
  8. UVa 11472 - Beautiful Numbers
  9. LA 3015 - Zeros and Ones (Dhaka04)
  10. LA 3136 - Fun Game (Beijing04)
  11. LA 4643 - Twenty Questions (Tokyo09)
  12. LA 4794 - Sharing Chocolate (World Finals Harbin10, discussed in this section)

- More Advanced DP
  1. **UVa 00473 - Raucous Rockers *** (the input constraint is not clear[10])
  2. UVa 00607 - Scheduling Lectures
  3. UVa 00882 - The Mailbox Manufacturer ... (s: (lo, hi, mailbox_left); try all)
  4. UVa 10069 - Distinct Subsequences (use Java BigInteger)
  5. UVa 10081 - Tight Words (use doubles)
  6. UVa 10163 - Storage Keepers (try all possible safe line L and run DP[11])
  7. UVa 10164 - Number Game (a bit number theory (modulo), backtracking[12])
  8. UVa 10271 - Chopsticks (details in footnote[13])
  9. **UVa 10482 - The Candyman Can *** (drop one parameter between a, b, or c)
  10. **UVa 10626 - Buying Coke *** (details in footnote[14])
  11. UVa 10898 - Combo Deal (similar to DP + bitmask; state can be stored as integer)
  12. UVa 11285 - Exchange Rates (maintain the best CAD and USD at any given day)
  13. LA 3404 - Atomic Car Race (Tokyo05)
  14. LA 3620 - Manhattan Wiring (Yokohama06)
  15. LA 3794 - Party at Hali-Bula (Tehran06, DP on Tree)
  16. LA 3797 - Bribing FIPA (Tehran06, DP on Tree)
  17. LA 4031 - Integer Transmission (Beijing07)
  18. LA 4106 - ACORN (Singapore07) (DP with dimension reduction)
  19. LA 4141 - Disjoint Paths (Jakarta08)
  20. LA 4143 - Free Parentheses (Jakarta08, problem author: Felix Halim)
  21. LA 4146 - ICPC Team Strategy (Jakarta08) (DP with 3 states)
  22. LA 4336 - Palindromic paths (Amritapuri08)
  23. LA 4337 - Pile it down (Amritapuri08, Game Theory, DP)
  24. LA 4526 - Inventory (Hsinchu09)
  25. LA 4712 - Airline Parking (Phuket09)
  26. LA 4791 - The Islands (World Finals Harbin10, Bitonic TSP)

8.5 Chapter Notes

This chapter and the second edition of this book end here. After writing so much, we become more aware of the presence of more interesting techniques that we are not yet able to cover in the second edition of this book. Expect this chapter to be improved substantially in the *third* edition.

To be continued... :)

There are ≈ **52 UVa (+ 31 others) programming exercises** discussed in this chapter.

There are **15 pages** in this chapter.

[10] Use `vector` and compact states.

[11] DP: s: id, N_left; t: hire/skip person 'id' for looking at K storage.

[12] Do memoization on DP state: (sum, taken).

[13] 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.

[14] DP state: (bought, n_5, n_{10}). We drop n_1 as it can be derived later. Dropping n_1 is the most beneficial one.

This page is intentionally left blank to keep the number of pages per chapter even.

Appendix A

Hints/Brief Solutions

In this appendix, you will see the hints or brief solutions for many – but not all[1] – mini exercises in this book. Please refrain from looking at the solution unless necessary.

Chapter 1

Exercise 1.2.1: The complete Table A.1 is shown below.

UVa	Title	Problem Type	Hint
10360	Rat Attack	Complete Search or Dynamic Programming	Section 3.2 or 3.5
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 + bitmasks (Non Classical)	Section 8.4.1
11635	Hotel Booking	Graph (Decomposition: Dijkstra's + BFS)	Section 8.2
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 8.2
11512	GATTACA	String (Suffix Array, LCP, LRS)	Section 6.6
10065	Useless Tile Packers	Geometry (Convex Hull + Area of Polygon)	Section 7.3.7

Table A.1: Exercise: Classify These UVa Problems

Exercise 1.2.2: The answers are:

1. (b) Use priority queue data structure (heap) (Section 2.2.2).
2. If list L is static: (b) Simple Array that is pre-processed with Dynamic Programming (Section 2.2.1 & 3.5). If list L is dynamic: then (f) Fenwick Tree is a better answer (easier to implement than Segment Tree).
3. (a) Yes, such complete search is possible (Section 3.2).
4. (a) Dynamic Programming (Section 3.5, 4.2.5, & 4.7.1).
5. (a) Sieve of Eratosthenes (Section 5.5.1).
6. (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).
7. (b) The naïve approach above will not work. Use KMP or Suffix Array (Section 6.4 or 6.6)!
8. (b) No, we must find another way. Find the Convex Hull of the N points first in $O(n \log n)$ (Section 7.3.7). Let the number of points in $CH(S) = k$. This $k << N$! Then, find the two furthest points by examining all pairs of points in the $CH(S)$ in $O(k^2)$.

[1]This book is used as an official lecture material in Steven's module @ NUS: CS3233 - 'Competitive Programming'. It is necessary to hide the answers for some exercises as they will be used as class assignments.

Exercise 1.2.3: The codes are shown below:

```
// Java code for question 1, 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));
} }

// C++ code for question 2, 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));
}

// Java code for question 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"
} }


// Java code for question 4
class Main {
  public static void main(String[] args) {
    String S = "line: a70 and z72 will be replaced, but aa24 and a872 will not";
    System.out.println(S.replaceAll("(^| )+[a-z][0-9][0-9]( |$)+", " *** "));
} }

// Java code for question 5, 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!
} }

// C++ code for question 6, 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("%d ", p[i]);
    printf("\n");
  }
  while (next_permutation(p, p + N));
}
```

```
// C++ code for question 7, 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");
} }
```

Exercise 1.2.4: Answers for situation judging are in bracket:

1. You receive a WA response for a very easy problem. What should you do?
   (a) Abandon this problem and do another. **(not ok, your team will lose out).**
   (b) Improve the performance of your solution. **(not useful).**
   (c) Create tricky test cases and find the bug. **(the most logical answer).**
   (d) (In ICPC): Ask another coder in your team to re-do this problem. **(this is a logical answer. this can work although your team will lose precious penalty time).**

2. You receive a TLE response for an your $O(N^3)$ solution. However, maximum N is just 100. What should you do?
   (a) Abandon this problem and do 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 $N \leq \approx 200$).**
   (c) Create tricky test cases and find the bug. **(this is the answer; maybe your program is accidentally trapped in an infinite loop in some test cases).**

3. Follow up question (see question 2 above): What if maximum N is 100.000?
   (If $N > 200$, you have no choice but to improve the performance of the algorithm or use a faster algorithm).

4. You receive an RTE response. Your code runs OK in your machine. What should you do?
   Possible causes for RTE are usually array size too small or stack overflow/infinite recursion. Design test cases that can possibly cause your code to end up with these situations.

5. 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 (your team) do?
   (a) Abandon the problem with WA code, switch to that other problem in attempt to solve one more problem. **(in individual contests like IOI, this may be a good idea).**
   (b) Insist that you have to debug the WA code. There is not enough time to start working on a new code. **(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' codes).**
   (c) (In ICPC): Print the WA code. Ask two other team members to scrutinize the printed code while one coder switches to that other problem in attempt to solve TWO more problems. **(if the idea for another problem is can be coded in less than 30 minutes, then code this one while hoping your team mates can find the bug for the WA code by looking at the printed code).**

Chapter 2

Exercise 2.2.2.1: Use C++ STL `set` (Java `TreeSet`) as it is a balanced BST that supports $O(\log n)$ insertions and deletions. We can use inorder traversal to print the data in the BST in sorted order (simply use C++ or Java `iterator`).

Exercise 2.2.2.2: Use C++ STL `map` (Java `TreeMap`) and a counter variable. Hash table is also possible but not necessary for programming contests. This trick is used quite frequently in various programming problems. Example usage:

```
char str[1000];
map<string, int> mapper;
int i, idx;

for (i = idx = 0; i < M; i++) {
  gets(str);
  if (mapper.find(str) != mapper.end()) mapper[str] = idx++;
}
```

Exercise 2.3.1.1: Straightforward. Solution is not shown.

Exercise 2.3.1.2: Solution is not shown.

Exercise 2.3.1.3: For Adjacency Matrix, do matrix transpose in $O(V^2)$. For Adjacency List, it is easier to construct a new Adjacency List with reversed edges in $O(V + E)$.

Exercise 2.3.2.1: For `int numDisjointSets()`, use additional integer counter `disjointSetsSize`. Initially, during `initSet(N)`, set `disjointSetsSize = N`. Then, during `unionSet(i, j)`, decrease `disjointSetsSize` by 1 if `!isSameSet(i, j)`. Finally, `int numDisjointSets()` can simply return the value of `disjointSetsSize`.

For `int sizeOfSet(int i)`, we use another `vi set_size(N)` initially set to all 1. Then, during `unionSet(i, j)`, do `set_size[findSet(j)] += set_size[findSet(i)]` if `!isSameSet(i, j)`. Finally, `int sizeOfSet(int i)` can simply return the value of `set_size[findSet(i)];`

These two variants have been implemented in the example code: `ch2_08_unionfind_ds.cpp`.

Exercise 2.3.2.2: Actually, the major aim of 'union by rank' heuristic is to keep the union-find tree as balanced as possible. This can be 'simulated' by occasionally calling `findSet(i)` after few union operations to 'compress the path'.

Exercise 2.3.3.1: Straightforward. Solution is not shown.

Exercise 2.3.3.2: No, using Segment Tree is overkill. There is a simple DP solution that uses $O(n)$ pre-processing and $O(1)$ per RSQ (see Section 3.5.2).

Exercise 2.3.3.3: Solution is not shown.

Exercise 2.3.4.1: The cumulative frequency is sorted, thus we can use *binary search*. Study the 'binary search the answer' technique discussed in Section 3.3. The time complexity is $O(\log^2 n)$.

Exercise 2.3.4.2: See TopCoder algorithm tutorial [17].

Chapter 3

Exercise 3.2.1.1: This is to avoid division operator so that we only work with integers!

Exercise 3.2.1.2: It is likely AC too as 10! $\approx$ 3 million.

Exercise 3.2.1.3: We can just check `fghij` from `01234` to `98765 / N`. This reduces the runtime further and ensure that `fghij` $\times$ `N` will only have 5 digits.

Exercise 3.2.1.4: Modify this code:

```
void backtrack(int col) {
  for (int tryrow = 1; tryrow <= 8; tryrow++) {          // try all possible row
    if (col == b && tryrow != a) continue;                      // add this line
    if (place(col, tryrow)) {     // if can place a queen at this col and row...
      row[col] = tryrow;                   // put this queen in this col and row
      if (col == 8) {        // a candidate solution and cell (a, b) has 1 queen
        printf("%2d      %d", ++lineCounter, row[1]);
        for (int j = 2; j <= 8; j++) printf(" %d", row[j]);
        printf("\n");
      }
      else
        backtrack(col + 1);                       // recursively try next column
} } }
```

Exercise 3.2.1.5: Straightforward. Solution is not shown.

Exercise 3.2.1.6: We must consider the two diagonals too. To further speed up the solution, use bitmask technique (lightweight set of Boolean data structure) discussed in Section 2.2.1.

Exercise 3.3.1.1: Solution is not shown.

Exercise 3.4.1.1: UVa 11389 is also a 'load balancing' problem. Sort input. Find the pattern.

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.

Exercise 3.5.1.2 and **Exercise 3.5.1.3**:

```
/* UVa 11450 - Wedding Shopping - Bottom Up - Row Major - Space Saving Trick */
#include <cstdio>
#include <cstring>
using namespace std;

int main() {
  int i, j, l, TC, M, C, cur, price[25][25];    // price[g (<=20)][model (<=20)]
  bool can_reach[2][210];                   // can_reach table[2][money (<=200)]

  scanf("%d", &TC);
  while (TC--) {
    scanf("%d %d", &M, &C);
    for (i = 0; i < C; i++) {
      scanf("%d", &price[i][0]);   // to simplify coding, store K in price[i][0]
      for (j = 1; j <= price[i][0]; j++) scanf("%d", &price[i][j]);
    }
    memset(can_reach, false, sizeof can_reach);             // clear everything
    cur = 1; // row 1 is the active row; row !cur (or row 0) is the previous row
    for (i = 1; i <= price[0][0]; i++)                        // initial values
      can_reach[!cur][M - price[0][i]] = true; // only using first garment g = 0
    for (j = 1; j < C; j++) {                       // for each remaining garment
      memset(can_reach[cur], false, sizeof can_reach[cur]);    // reset this row
      for (i = 0; i < M; i++) if (can_reach[!cur][i]) // if can reach this state
        for (l = 1; l <= price[j][0]; l++)
          if (i - price[j][l] >= 0)                             // flag the rest
            can_reach[cur][i - price[j][l]] = true; // as long as it is feasible
      cur = !cur;                                   // key trick: swap active row
    }
```

```
    for (i = 0; i <= M && !can_reach[!cur][i]; i++);   // answer in the last row
    if (i == M + 1) printf("no solution\n");        // no '1' in the last column
    else            printf("%d\n", M - i);
 } } // return 0;
```

Exercise 3.5.2.1: Study Jay Kadane's algorithm. The main idea is to reset the running sum to 0 if the current running sum goes down to negative. If the 2D version looks hard, try the 1D version in **Exercise 3.5.2.3**.

Exercise 3.5.2.2: Solution is not shown.

Exercise 3.5.2.3: See the similar idea used in Fenwick's Tree (Section 2.3.4).

Exercise 3.5.2.4: Solution is not shown.

Chapter 4

Exercise 4.2.2.1: Plain integer array can be used if we know the number of vertices in the graph (V) and the vertices can be indexed easily, otherwise C++ STL `map` is good especially if BFS is done on 'state search graph'. Actually, the solution for this exercise has been shown in our Dijkstra's implementation that uses distance vector implementation. Note that if we do not need the distance information, we can just use a Boolean array to mark a vertex as visited or not instead of `map` and make the routine slightly more efficient.

Exercise 4.2.2.2: Adjacency Matrix requires $O(V)$ to enumerate list of neighbors of a vertex whereas Adjacency List just requires $O(k)$ where k is the number of actual neighbors of a vertex.

Exercise 4.2.3.1: Initially, all vertices are disjoint. For every `edge(u, v)`, do `unionSet(u, v)`. The state of disjoint sets after processing all edges represent the connected components. BFS solution is 'trivial' and is not shown.

Exercise 4.2.5.1: This is a post-order traversal. Function `dfs2` visits all the children of u before appending vertex u at the back of toposort vector. This satisfies the topological sort property!

Exercise 4.2.5.2: To get your implementation Accepted for UVa 11060 - Beverages, you have to use `priority_queue` instead of `queue`.

Exercise 4.2.6.1: Solution is not shown.

Exercise 4.2.6.2: Yes.

Exercise 4.2.7.1: Two back edges: $2 \rightarrow 1$ and $6 \rightarrow 4$.

Exercise 4.2.8.1: Articulation points: 1, 3, Bridges: 0-1, 3-4, 6-7.

Exercise 4.2.9.1: In our opinion, Tarjan's version is simpler as it links well with our other DFS variants in Section 4.2.1. Kosaraju's version involves graph transpose that is mentioned briefly in Section 2.3.1.

Exercise 4.2.9.2: Yes.

Exercise 4.2.9.3: Solution is not shown.

Exercise 4.3.2.1: We can stop when the number of disjoint sets is already one. The modification is simple. 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 4.3.4.2: First, find the 'Maximum' ST of the given weighted graph.

Exercise 4.4.2.1: For this variant, the solution is easy. Create a super source that is connected to all sources with edge weight 0. Then enqueue the super source. Better still, simply enqueue all the sources before running the BFS loop. As this is just one BFS call, it runs in $O(V + E)$.

Exercise 4.4.3.1: On positive weighted graph, yes. On graph with negative weight edges, it runs slower but still correct (unlike the original Dijkstra's).

Exercise 4.4.3.2: Use `set<ii>`. This set stores sorted pair of vertex information as shown in Section 4.4.3. Vertex with 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.

Exercise 4.4.4.1: Solution is not shown.

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 Floyd Warshall's directly on $V \leq 1000$ will result in TLE. Use the property of SCC.

Exercise 4.5.3.2: Straightforward. Solution is not shown.

Exercise 4.5.3.3: Replace addition with multiplication. Check if the main diagonal > 1.0.

Exercise 4.6.3.1: A. 150; B = 125; C = 60.

Exercise 4.6.3.2: Use *both* Adjacency List (for fast enumeration of neighbors) and Adjacency Matrix (for fast access to edge weight/residual capacity) of the same flow graph. Concentrate on improving this line: `for (int v = 0; v < MAX_V; v++)`. We can also replace `vi dist(MAX_V, INF);` to `bitset<MAX_V> visited` to speed up the code a little bit more.

Exercise 4.6.3.3: Solution is not shown.

Exercise 4.6.3.4: Solution is not shown.

Exercise 4.6.4.1: Solution is not shown.

Exercise 4.7.1.1: Solution is not shown.

Exercise 4.7.2.1: The better solution uses Lowest Common Ancestor (LCA).

Exercise 4.7.4.1: Solution is not shown.

Chapter 5

Exercise 5.2.1: <cmath> in C/C++ has `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 few zeroes are added after a multiplication from $n!$ to $(n + 1)!$).

Exercise 5.3.1.2: It is possible, list 25! as its prime factors see if there are one factor 7 (yes), and three factors 11 (unfortunately no). So 25! is not divisible by 9317 (7×11^3). Alternative approach: Use modulo arithmetic.

Exercise 5.3.2.1: For base number conversion of integers that fit in `int` data type, we can use `Integer.parseInt(String s, int radix)` and `Integer.toString(int i, int radix)` in Java **Integer** class. This one is slightly faster.

Exercise 5.3.2.2:
Study `http://download.oracle.com/javase/1,5.0/docs/api/java/math/BigDecimal.html`.

Exercise 5.4.1.1: Actually $fib(n) = ((\phi^n) - (1/\phi^n))/\sqrt{5}$ (Binet's formula). Mathematically, it should be correct for larger n. But since double precision data type is limited, we have discrepancies for larger n. Anyway, this closed form formula is correct up to $fib(69)$ if implemented using typical double data type in a computer program.

Exercise 5.4.2.1: $C(n,2) = \frac{n!}{(n-2)!\times 2!} = \frac{n\times(n-1)\times(n-2)!}{(n-2)!\times 2} = \frac{n\times(n-1)}{2} = O(n^2)$.

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 the fundamental counting principle above. The answer for this exercise 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: $\frac{n!}{(n_1)!\times(n_2)!\times\ldots\times(n_k)!}$ where n_i is the frequency of each unique letter i and $n_1 + n_2 + \ldots + n_k = n$. 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 5.4.4.6: Do this on your own.

Exercise 5.5.4.1: Since the largest prime in `vi 'primes'` is 9999991, this code can therefore handles $N \leq 9999991^2 = 99999820000081$. If the smallest prime factor of N is greater than 9999991, for example, $N = 1010189899^2 = 1020483632041630201$, 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 reckon there is no prime factoring algorithm that is fast enough under 1 or 3 seconds of programming contest setting.

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 (N != 1 && (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 (N != 1 && (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: Statement 2 and 4 are not valid. The other 3 are valid.

Chapter 6

Exercise 6.2.1: In C, a string is stored as an array of characters terminated by null, for example `char str[30x10 + 10], line[30 + 10];`). It is a good practice to declare array size slightly bigger than requirement to avoid "off by one" bug. To read the input line by line and then concatenate them, we can first set `strcpy(str, ‘‘’’);`, then use `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. Then, we can combine the lines into a longer string using `strcat(str, line);`. We append a space so that the last word from one line is not accidentally combined with the first word of the next line. We keep repeating this process until `strncmp(line, ‘‘.......’’, 7) == 0`.

Exercise 6.2.2: For finding a substring in a relatively short string (i.e. the standard string matching problem), we can just use library function. In C, we can use `p = strstr(str + pos, substr);`. `p == NULL` if `substr` is not found in `str + pos`. If there are multiple copies of `substr` in `str`, we can set the value of `pos` to be the index of the first occurrence of `substr` plus one so that we can get the second occurrence, and so on. Note: This 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 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)` (and `isdigit(ch)`) to check whether a given character is alphabet `[A-Za-z]` (digit). To test whether a character is a vowel, one method is to prepare a string `vowel = "abcde";` 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 6.2.4-5: One of the easiest way to tokenize a string is to use `strtok(str, delimiters);` in C. These tokens can be stored in C++ `vector<string> tokens`. We can then 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()`.

Exercise 6.2.6: We can use C++ STL `map<string, int>` to keep track the frequency of each word. Every time we encounter a new token, increase the corresponding frequency by one. Finally, scan through all tokens and determine the one with the highest frequency.

Exercise 6.2.7: Read char by char 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 setter can set a ridiculously long string to break your code.

Exercise 6.4.1 and **Exercise 6.4.2**: Run our sample code.

Exercise 6.5.1.1: Different scoring scheme will yield different (global) alignment. If given 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.

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 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 solution. So if the actual solution is needed, we cannot use this space saving trick.

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: Solution is not shown. 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, 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 6.6.2.1: Straightforward. Solution is not shown.

Exercise 6.6.3.1: 'CA' is found, 'CAT' is not.

Exercise 6.6.3.2: 'ACATTA'.

Exercise 6.6.3.3: 'EVEN'.

Exercise 6.6.3.4: 'EVE'. Details not shown.

Exercise 6.6.3.5: 'EVEN' again. Details not shown.

Exercise 6.6.5.1: The answer is the same as **Exercise 6.6.3.2** which is 'ACATTA'.

Exercise 6.6.5.2: The answer is the same as **Exercise 6.6.3.3** which is 'EVEN'.

Chapter 7

Exercise 7.2.1.1: 3.0.

Exercise 7.2.1.2: (-3.0, 4.0).

Exercise 7.2.2.1: The line equation $y = mx + c$ cannot handle all the cases. Vertical lines has 'infinite' gradient/slope in this equation. 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 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.

```
struct line2 { double m, c; };                  // another way to represent a line
```

```
int points_to_line(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). This result is different from the previous **Exercise 7.2.2.8**. 'Translate then Rotate' is different from 'Rotate then translate'. Be careful in sequencing them.

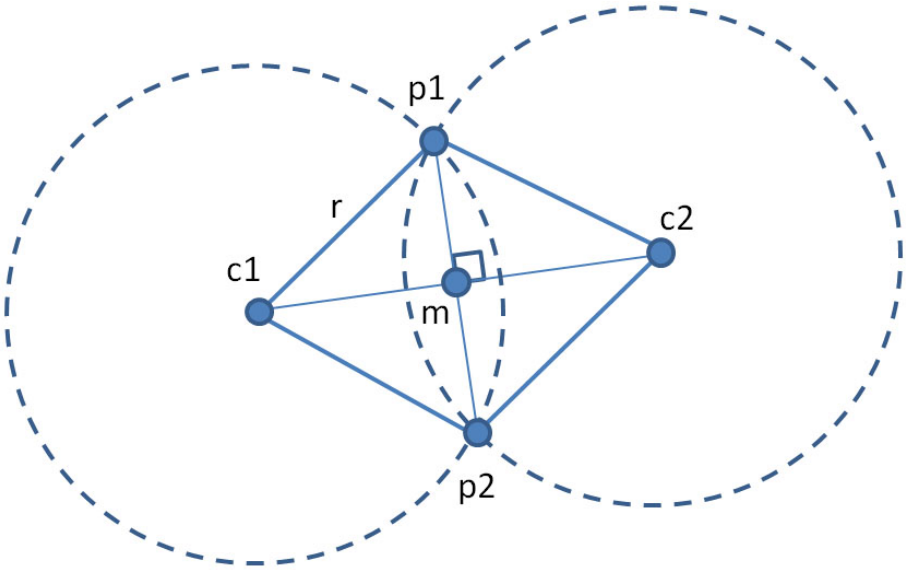
Exercise 7.2.2.10: The solution is shown below:

```
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 7.2.3.1: See the figure in the next page. 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: To avoid 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 7.2.4.2: Straightforward. Solution is not shown

Exercise 7.3.3.1: Test it by yourself. Solution is not shown.

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: Check your programming language manual.

Exercise 7.3.7.3: Solution is not shown.

Chapter 8

Exercise 8.2.1: Solution is not shown.

Exercise 8.4.1.1: As we want a perfect matching, we do not want an unmatched vertex! So if we already get an optimal matching for vertex p1 over all possible choices of p2, then we can break. This will reduce the time complexity from $O((2N)^2 \times 2^{2N})$ down to $O((2N) \times 2^{2N})$".

Exercise 8.4.5.1: Solution is not shown.

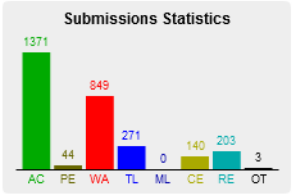
Appendix B

uHunt

uHunt (`http://felix-halim.net/uva/hunting.php`) is a web-based tool created by one of the author of this book (Felix Halim) to complement the UVa online judge [28]. This tool is created to help UVa users in keeping track which problems that he/she has (and has not) solved. Considering that UVa has ≈ 2950 problems, today's programmers indeed need a tool like this (see Figure B.1).

Last Submissions (auto update) **[UVa Link]** Show : 5 | 10 | 50 | 100 | 500 | ALL

Problem		Verdict	Lang	Time	Best	Rank	Submit Time
10653 - Bombs! NO they are	discuss	Accepted	C++	0.592	0.124	167	9 days ago
10653 - Bombs! NO they are	discuss	Accepted	C++	0.528	0.124	147	9 days ago
10653 - Bombs! NO they are	discuss	Compile error	C++	0.000	0.124	-	9 days ago
10107 - What is the Median?	discuss	Accepted	C++	0.192	0.000	1595	12 days ago
100 - The 3n + 1 problem	discuss	Accepted	C++	0.716	0.000	12532	13 days ago



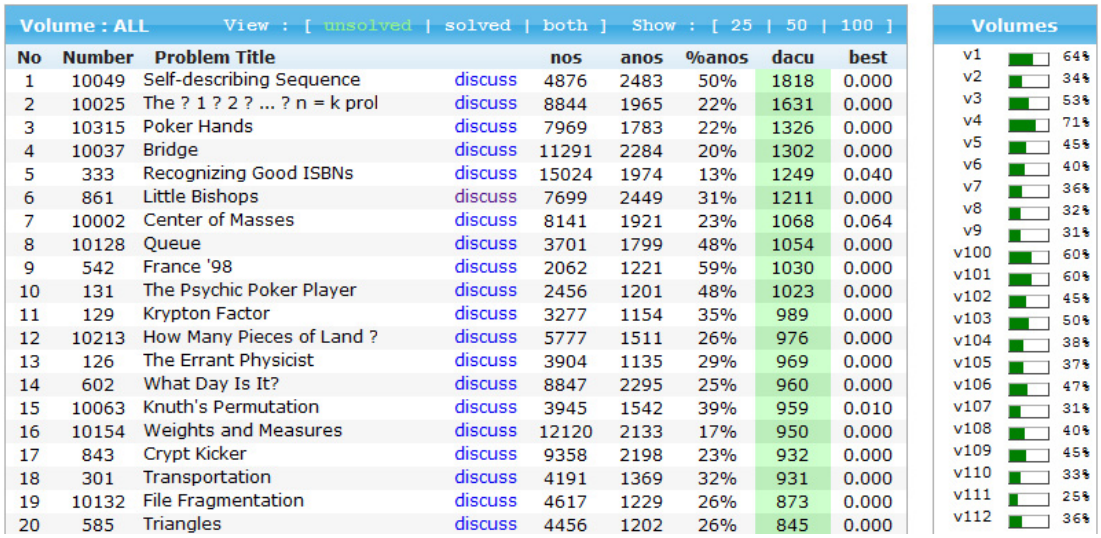
Solved : 1180, Submissions : 2883

100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 127 128 130
133 136 138 139 140 141 143 144 145 146 147 148 151 152 153 154 155 156 160 161 162 164 165 166 167 168 170 184 186
187 188 190 191 193 195 200 201 202 208 213 216 218 220 222 227 231 232 245 247 253 255 256 259 260 263 264 270 271
272 275 276 278 280 290 291 294 296 297 299 300 305 306 311 314 315 320 321 324 325 326 327 331 332 334 335 336 337
339 340 341 343 344 346 347 348 349 350 352 353 355 356 357 361 362 369 371 374 375 377 378 379 380 381 382 383 384
386 389 391 392 394 397 400 401 402 403 405 406 408 409 410 412 413 414 416 417 422 423 424 429 433 434 435 436 437
438 439 440 441 442 443 444 445 446 448 450 452 454 455 457 458 459 460 462 464 465 466 467 468 469 471 473 474 476
477 478 481 482 483 484 485 486 487 488 489 490 492 494 495 496 497 498 499 501 507 514 516 524 526 530 531 532 534
535 536 537 538 539 540 541 543 544 545 547 551 555 556 558 562 565 567 568 571 572 573 574 575 576 579 580 583 584
587 590 591 594 598 599 608 610 612 614 615 616 617 619 620 621 622 623 624 626 627 628 630 634 636 637 639 640 641
642 644 647 652 657 661 670 671 673 674 677 679 681 686 694 696 699 700 701 703 706 712 713 714 719 725 727 729 732
735 737 739 740 741 743 748 750 753 755 756 759 760 762 763 776 782 784 785 787 789 793 795 796 808 811 815 820 821
824 825 833 834 836 837 839 846 847 850 852 855 856 858 860 865 868 869 871 872 880 882 884 892 893 895 897 900 902 906 907 908 910 913 914 920 924 926
927 928 929 932 933 941 944 948 957 962 963 967 974 978 983 986 988 990 991 993 10000 10003 10004 10005 10006 10007 10008 10009 10010 10012 10013 10014
10015 10016 10017 10018 10019 10020 10026 10033 10034 10035 10036 10038 10041 10042 10044 10047 10048 10050 10051 10054 10055 10056 10058 10060
10061 10062 10065 10066 10067 10069 10070 10071 10074 10075 10077 10078 10079 10080 10081 10082 10083 10088 10090 10092 10093 10094 10098 10099
10100 10101 10102 10104 10105 10106 10107 10110 10111 10112 10113 10114 10115 10116 10125 10127 10129 10130 10131 10136 10137 10139 10140 >> more

Figure B.1: Steven's statistics as of 1 August 2011

The original version of uHunt (that stands for UVa hunting) is to help UVa users in finding the 20 easiest problems to solve next. 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 ≈ 115266 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 B.2).

Today's uHunt is much more flexible than its predecessors as it also allows users to rank problems that they *already solve* based on the runtime difference of their AC code versus the best AC code. A (wide) gap 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.



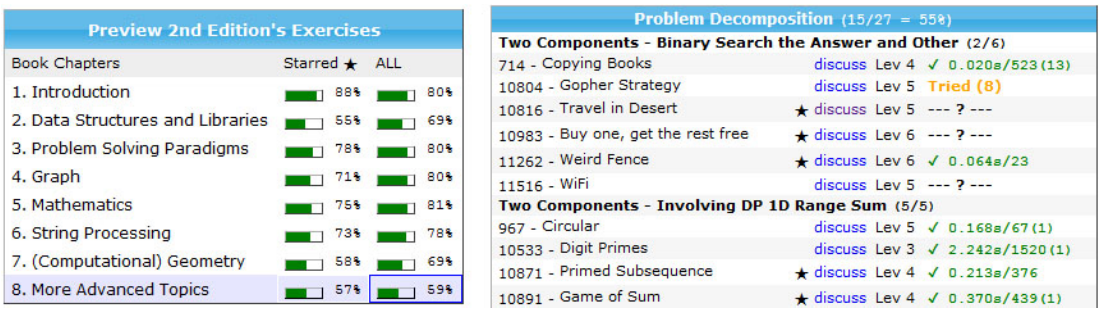
Volume : ALL View : [unsolved | solved | both] Show : [25 | 50 | 100]

No	Number	Problem Title		nos	anos	%anos	dacu	best
1	10049	Self-describing Sequence	discuss	4876	2483	50%	1818	0.000
2	10025	The ? 1 ? 2 ? ... ? n = k prol	discuss	8844	1965	22%	1631	0.000
3	10315	Poker Hands	discuss	7969	1783	22%	1326	0.000
4	10037	Bridge	discuss	11291	2284	20%	1302	0.000
5	333	Recognizing Good ISBNs	discuss	15024	1974	13%	1249	0.040
6	861	Little Bishops	discuss	7699	2449	31%	1211	0.000
7	10002	Center of Masses	discuss	8141	1921	23%	1068	0.064
8	10128	Queue	discuss	3701	1799	48%	1054	0.000
9	542	France '98	discuss	2062	1221	59%	1030	0.000
10	131	The Psychic Poker Player	discuss	2456	1201	48%	1023	0.000
11	129	Krypton Factor	discuss	3277	1154	35%	989	0.000
12	10213	How Many Pieces of Land ?	discuss	5777	1511	26%	976	0.000
13	126	The Errant Physicist	discuss	3904	1135	29%	969	0.000
14	602	What Day Is It?	discuss	8847	2295	25%	960	0.000
15	10063	Knuth's Permutation	discuss	3945	1542	39%	959	0.010
16	10154	Weights and Measures	discuss	12120	2133	17%	950	0.000
17	843	Crypt Kicker	discuss	9358	2198	23%	932	0.000
18	301	Transportation	discuss	4191	1369	32%	931	0.000
19	10132	File Fragmentation	discuss	4617	1229	26%	873	0.000
20	585	Triangles	discuss	4456	1202	26%	845	0.000

Volumes: v1 64% | v2 34% | v3 53% | v4 71% | v5 45% | v6 40% | v7 36% | v8 32% | v9 31% | v100 60% | v101 60% | v102 45% | v103 50% | v104 38% | v105 37% | v106 47% | v107 31% | v108 40% | v109 45% | v110 33% | v111 25% | v112 36%

Figure B.2: Hunting the next easiest problems using 'dacu'

Another new feature since 2010 is the integration of the ≈ 1198 programming exercises from this book (see Figure B.3). 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).



Preview 2nd Edition's Exercises

Book Chapters	Starred ★	ALL
1. Introduction	88%	80%
2. Data Structures and Libraries	55%	69%
3. Problem Solving Paradigms	78%	80%
4. Graph	71%	80%
5. Mathematics	75%	81%
6. String Processing	73%	78%
7. (Computational) Geometry	58%	69%
8. More Advanced Topics	57%	59%

Problem Decomposition (15/27 = 55%)

Two Components - Binary Search the Answer and Other (2/6)

714 - Copying Books	discuss	Lev 4	✓ 0.020s/523(13)
10804 - Gopher Strategy	discuss	Lev 5	Tried (8)
10816 - Travel in Desert	★ discuss	Lev 5	--- ? ---
10983 - Buy one, get the rest free	★ discuss	Lev 6	--- ? ---
11262 - Weird Fence	★ discuss	Lev 6	✓ 0.064s/23
11516 - WiFi	discuss	Lev 5	--- ? ---

Two Components - Involving DP 1D Range Sum (5/5)

967 - Circular	discuss	Lev 5	✓ 0.168s/67(1)
10533 - Digit Primes	discuss	Lev 3	✓ 2.242s/1520(1)
10871 - Primed Subsequence	★ discuss	Lev 4	✓ 0.213s/376
10891 - Game of Sum	★ discuss	Lev 4	✓ 0.370s/439(1)

Figure B.3: The programming exercises in this book are integrated in uHunt

Building a web-based tool like uHunt is a computational challenge. There are over ≈ 9099867 submissions from ≈ 115266 users (≈ 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 [16], Fenwick Tree, data compression, etc.

We ourselves are using this tool extensively, as can be seen in Figure B.4. 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 and Steven's intensive problem solving activity in the past two years (late 2009-present) to prepare this book.

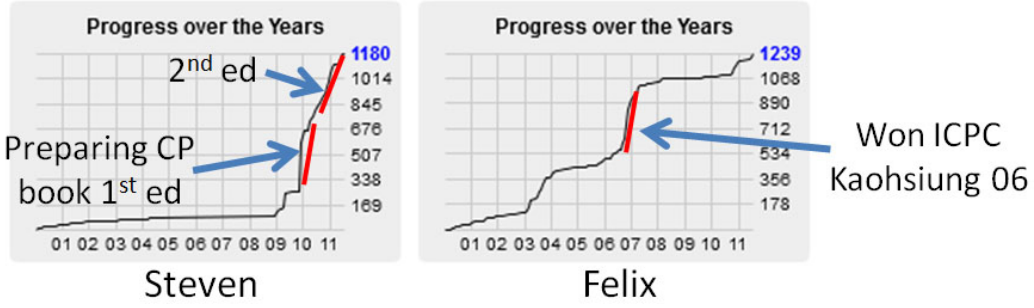


Figure B.4: Steven's & Felix's progress in UVa online judge (2000-present)

Appendix C

Credits

The problems discussed in this book are mainly taken from UVa online judge [28], ACM ICPC Live Archive [20], and past IOI tasks (mainly from 2009-2010). So far, we have contacted the following authors to get their permissions (in alphabetical order): Brian C. Dean, Colin Tan Keng Yan, Derek Kisman, Gordon V. Cormack, Jane Alam Jan, Jim Knisely, Jittat Fakcharoenphol, Manzurur Rahman Khan, Melvin Zhang Zhiyong, Michal Forišek, Mohammad Mahmudur Rahman, Piotr Rudnicki, Rob Kolstad, Rujia Liu, Shahriar Manzoor, Sohel Hafiz, and 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 ($\approx$ 1198) 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`

Appendix D

Plan for the Third Edition

The first edition of this book is released on 9 August 2010 and the second edition one year later on 1 August 2011. Is this going to be a yearly trend?

We all have to admit that the world of competitive programming is constantly evolving and the programming contest problems are getting harder. To avoid getting *too* outdated, it is natural for a book like this one to have incremental updates (anyway, a book is always out of date by the time it goes to print). Since we believe that a good book is the one that keeps being updated by it's authors, we do have plan to release the third edition in the future, but not in 2012.

The gap between the first and the second edition is 'small' (only one year) as there are so much to write if one writes a book from scratch. We believe that the second edition is already a significant improvement over the first edition. This makes the expectation for the third edition even higher. To have a good quality third edition, we plan to release the third edition only in two or three years time (that means around 2013 or 2014). So, the second edition edition of this book has about two to three years active life in the competitive programming world.

These are our big plans to prepare the third edition in two or three years time:

- Deal with the errata (hopefully minimal) that are found after the release of second edition. Online errata is at: `https://sites.google.com/site/stevenhalim/home/third`
- We will keep improving our problem solving skills and solve many more programming problems. By the time we are ready to publish the third edition, we expect the number of problems that we (Steven and Felix) have solved (combined) is around 2000, instead of 1502 as of now ($\approx$ 1198 of which are discussed). We will then share what we learn from those problems.
- Make the existing Chapter 1-7 a little bit more user friendly to reach newer competitive programmers: more examples, more illustrations, more conceptual exercises.
- Add discussion of most data structures and algorithms that are currently only mentioned in the chapter notes.
- Expand Chapter 8 (substantially) to serve the needs of today's competitive programmers who are already (or trying to) master the content of the second edition of this book.

We need your feedbacks

You, the reader, can help us improve the quality of the third edition of this book. If you spot any technical, grammatical, spelling errors, etc in this book or if you want to contribute certain parts for the third edition of this book (i.e. I have a better example/algorithm to illustrate a certain point), etc, please send an email to the main author directly: stevenhalim@gmail.com.

Bibliography

[1] Ahmed Shamsul Arefin. *Art of Programming Contest (from Steven's old Website)*. Gyankosh Prokashoni (Available Online), 2006.

[2] Frank Carrano. *Data Abstraction & Problem Solving with C++*. Pearson, 5th edition, 2006.

[3] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Cliff Stein. *Introduction to Algorithm*. MIT Press, 2nd edition, 2001.

[4] Sanjoy Dasgupta, Christos Papadimitriou, and U Vazirani. *Algorithms*. McGraw Hill, 2008.

[5] Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Cheong Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer, 2nd edition, 2000.

[6] Jack Edmonds. Paths, trees, and flowers. *Canadian Journal on Maths*, 17:449–467, 1965.

[7] Fabian Ernst, Jeroen Moelands, and Seppo Pieterse. Teamwork in Prog Contests: 3 * 1 = 4. http://xrds.acm.org/article.cfm?aid=332139.

[8] Project Euler. Project Euler. http://projecteuler.net/.

[9] Peter M. Fenwick. A New Data Structure for Cumulative Frequency Tables. *Software: Practice and Experience*, 24 (3):327–336, 1994.

[10] Michal Forišek. IOI Syllabus. http://people.ksp.sk/ misof/ioi-syllabus/ioi-syllabus-2009.pdf.

[11] Michal Forišek. The difficulty of programming contests increases. In *International Conference on Informatics in Secondary Schools*, 2010.

[12] Felix Halim, Roland Hock Chuan Yap, and Yongzheng Wu. A MapReduce-Based Maximum-Flow Algorithm for Large Small-World Network Graphs. In *ICDCS*, 2011.

[13] Steven Halim and Felix Halim. Competitive Programming in National University of Singapore. Ediciones Sello Editorial S.L. (Presented at Collaborative Learning Initiative Symposium CLIS @ ACM ICPC World Final 2010, Harbin, China, 2010.

[14] 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.

[15] 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.

[16] Stratos Idreos. *Database Cracking: Towards Auto-tuning Database Kernels*. PhD thesis, CWI and University of Amsterdam, 2010.

[17] TopCoder Inc. Algorithm Tutorials. http://www.topcoder.com/tc?d1=tutorials&d2=alg_index&module=Static.

[18] TopCoder Inc. PrimePairs. Copyright 2009 TopCoder, Inc. All rights reserved. http://www.topcoder.com/stat?c=problem_statement&pm=10187&rd=13742.

[19] TopCoder Inc. Single Round Match (SRM). http://www.topcoder.com/tc.

[20] Competitive Learning Institute. ACM ICPC Live Archive. http://livearchive.onlinejudge.org/.

[21] IOI. International Olympiad in Informatics. http://ioinformatics.org.

[22] Juha Kärkkäinen, Giovanni Manzini, and Simon J. Puglisi. Permuted Longest-Common-Prefix Array. In *CPM, LNCS 5577*, pages 181–192, 2009.

[23] Jon Kleinberg and Eva Tardos. *Algorithm Design*. Addison Wesley, 2006.

[24] Anany Levitin. *Introduction to The Design & Analysis of Algorithms*. Addison Wesley, 2002.

[25] Rujia Liu. *Algorithm Contests for Beginners (In Chinese)*. Tsinghua University Press, 2009.

[26] Rujia Liu and Liang Huang. *The Art of Algorithms and Programming Contests (In Chinese)*. Tsinghua University Press, 2003.

[27] Institute of Mathematics and Lithuania Informatics. Olympiads in Informatics. http://www.mii.lt/olympiads_in_informatics/.

[28] University of Valladolid. Online Judge. http://uva.onlinejudge.org.

[29] USA Computing Olympiad. USACO Training Program Gateway. http://train.usaco.org/usacogate.

[30] Joseph O'Rourke. *Computational Geometry in C*. Cambridge U Press, 2nd edition, 1998.

[31] Kenneth H. Rosen. *Elementary Number Theory and its applications*. Addison Wesley Longman, 4th edition, 2000.

[32] Robert Sedgewick. *Algorithms in C++, Part 1-5*. Addison Wesley, 3rd edition, 2002.

[33] Steven S Skiena. *The Algorithm Design Manual*. Springer, 2008.

[34] Steven S. Skiena and Miguel A. Revilla. *Programming Challenges*. Springer, 2003.

[35] SPOJ. Sphere Online Judge. http://www.spoj.pl/.

[36] Wing-Kin Sung. *Algorithms in Bioinformatics: A Practical Introduction*. CRC Press (Taylor & Francis Group), 1st edition, 2010.

[37] Esko Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14 (3):249–260, 1995.

[38] Baylor University. ACM International Collegiate Programming Contest. http://icpc.baylor.edu/icpc.

[39] Tom Verhoeff. 20 Years of IOI Competition Tasks. *Olympiads in Informatics*, 3:149166, 2009.

[40] Adrian Vladu and Cosmin Negruşeri. Suffix arrays - a programming contest approach. 2008.

[41] Henry S Warren. *Hacker's Delight*. Pearson, 1st edition, 2003.

[42] Wikipedia. The Free Encyclopedia. http://en.wikipedia.org.

Index