Contents

Acknowledgements

.

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.
- 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 CS3233 students in Sem2 AY2009/2010 who help me verify the content of this book plus the Live Archive contribution.
- My friend Ilham Winata Kurnia for proof reading the manuscript.

Copyright

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" [23], 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
UVa Online Judge site creator
ACM-ICPC International Steering Committee Member and Problem Archivist
University of Valladolid
http://uva.onlinejudge.org
http://acmicpc-live-archive.uva.es

Preface

This is a book that every competitive programmer must read – and master, at least during the middle phase of their programming career: when they want to leap forward from 'just knowing some programming language commands' and 'some algorithms' to become a top programmer.

Typical readers of this book will be: 1). Thousands University students competing in annual ACM International Collegiate Programming Contest (ICPC) [27] regional contests, 2). Hundreds Secondary or High School Students competing in annual International Olympiad in Informatics (IOI) [12], 3). Their coaches who are looking for a comprehensive training materials [9], and 4). Basically anyone who loves problem solving using computer.

Beware that this book is *not* for a novice programmer. When we wrote the book, we set it for readers who have knowledge in basic programming methodology, familiar with at least one programming language (C/C++/Java), and have passed basic data structures and algorithms (or equivalent) typically taught in year one of Computer Science University curriculum.

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

Use `uva.onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&category=118`, `felix-halim.net/uva/hunting.php`, `www.uvatoolkit.com/problemssolve.php`, and `www.comp.nus.edu.sg/~stevenha/programming/acmoj.html` to help you to deal with UVa [17] problems listed in this book.

We know that one probably cannot win an ACM ICPC regional or get a gold medal in IOI just by mastering the *current version* of this book. While we have included a lot of material in this book, we are well aware that much more than what this book can offer, are required to achieve that feat. Some pointers are listed throughout this book for those who are hungry for more.

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 year 2010 and beyond, the level of competitions in ICPC and IOI increase because many of the contestants have mastered the content of this book. We 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 no chance of doing well in future ICPCs and IOIs. If such increase in 'required lowerbound knowledge' happens, this book has fulfilled its objective of advancing the level of human knowledge in this era.

To a better future of humankind,
Steven and Felix Halim

PS: To obtain example source codes, visit `http://sites.google.com/site/stevenhalim`.
To obtain PowerPoint slides/other instructional materials (only for coaches), send a personal request email to stevenhalim@gmail.com.

Authors' Profiles

Steven Halim, PhD <stevenhalim@gmail.com>

Steven Halim is currently an instructor 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 silver and two bronze IOI medallists (2009).

Felix Halim, PhD Candidate <felix.halim@gmail.com>

Felix Halim is currently a PhD student in the same University: SoC, NUS. In terms of programming contests, Felix has much colorful reputation than his older brother. He was IOI 2002 contestant. His 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, **felix_halim** actively joins TopCoder Single Round Matches and his highest rating is a **yellow** coder.

Convention

There are a lot of C++ codes shown in this book. Many of them uses 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.

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

// Shortcuts for "common" data types in contests
typedef long long             ll;
typedef vector<int>           vi;
typedef pair<int, int>        ii;
typedef vector<ii>            vii;
typedef set<int>              si;
typedef map<string, int>      msi;

// To simplify repetitions/loops, Note: define your loop style and stick with it!
#define REP(i, a, b) \
  for (int i = int(a); i <= int(b); i++) // a to b, and variable i is local!
#define TRvi(c, it) \
  for (vi::iterator it = (c).begin(); it != (c).end(); it++)
#define TRvii(c, it) \
  for (vii::iterator it = (c).begin(); it != (c).end(); it++)
#define TRmsi(c, it) \
  for (msi::iterator it = (c).begin(); it != (c).end(); it++)

#define INF 2000000000 // 2 billion
// If you need to recall how to use memset:
#define MEMSET_INF 127 // about 2B
#define MEMSET_HALF_INF 63 // about 1B
//memset(dist, MEMSET_INF, sizeof dist); // useful to initialize shortest path distances
//memset(dp_memo, -1, sizeof dp_memo); // useful to initialize DP memoization table
//memset(arr, 0, sizeof arr); // useful to clear array of integers
```

Abbreviations

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
BST : Binary Search Tree

CC : Coin Change
CCW : Counter ClockWise
CS : Computer Science

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

ED : Edit Distance

GCD : Greatest Common Divisor

ICPC : International Collegiate Programming Contest
IOI : International Olympiad in Informatics

LA : Live Archive [11]
LCM : Lowest Common Ancestor
LCM : Lowest Common Multiple
LCS : Longest Common Subsequence
LIS : Longest Increasing Subsequence

MCM : Matrix Chain Multiplication
MCMF : Min-Cost Max-Flow
MLE : Memory Limit Exceeded
MST : Minimum Spanning Tree
MWIS : Maximum Weighted Independent Set

OJ : Online Judge

PE : Presentation Error

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

SSSP : Single-Source Shortest Paths
STL : Standard Template Library

TLE : Time Limit Exceeded

UVa : University of Valladolid [17]

WA : Wrong Answer
WF : World Finals

List of Tables

List of Figures

Chapter 1

Introduction

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

In this chapter, we introduce readers to the world of competitive programming. Hopefully you enjoy the ride and continue reading and learning until the very last page of this book, enthusiastically.

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 setter) have solved these problems before. 'Solve them' implies that we 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 input 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-rounded 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) [27] 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 recent ICPCs and International Olympiad in Informatics (IOI).

Illustration on solving UVa Online Judge [17] Problem Number 10911 (Forming Quiz Teams).

Abridged problem description: Let (x,y) be the coordinate of a student's house on a 2-D 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 $\sum_{i=1}^{N} d_i$ is **minimized**. Constraints: $N \leq 8; 0 \leq x, y \leq 1000$. Think first, try not to flip this page immediately!

Now, ask yourself, which one is you? Note that if you are unclear with the materials 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: Read the problem... confused @-@, never see this kind of problem before.
  Step 2: Try to code something... starting from reading non-trivial input and output.
  Step 3: Realize 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.
- (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 **'Edmond's Blossom Shrinking'** [34].
  But since the input size is small, this problem is solve-able using Dynamic Programming!
  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 do all those steps above in less than 30 minutes.
- Very Competitive programmer E:
  Of course, a very competitive programmer (e.g. the red 'target' coders in TopCoder [26]) may solve this 'classical' problem in less than 15 minutes...

1.2 Tips to be Competitive

If you strive to be like competitive programmer D or E in the illustration above. That is, you want to do well to qualify and get a medal in IOI [12]; to qualify in ACM ICPC [27] national, regional, and up to world final; or in other programming contests, then this book is definitely for you!

In subsequent chapters, you will learn basic to medium data structures and algorithms frequently appearing in recent programming contests, compiled from many sources [19, 6, 20, 2, 4, 14, 21, 16, 23, 1, 13, 5, 22, 15, 47, 24] (see Figure 1.5). But you will not just learn the algorithm, but also how to implement them efficiently and apply them to appropriate contest problem.

On top of that, you will also learn many tiny bits of programming tips from our experience that can be helpful in contest situation. We will start by giving you few general tips below:

Tip 0: Type Code Faster!

No kidding! Although this tip may not mean much as ICPC nor IOI are about typing speed competition, but 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 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 position of frequently used programming language characters, e.g. braces {} or () or <>, semicolon ';', single quote for 'char' and double quotes for "string", etc.

As a little practice, try typing this C++ code (a UVa 10911 solution above) as fast as possible.

```
/* Forming Quiz Teams. This DP solution will be explained in Section 3.4 */
#include <iostream>
#include <algorithm>
#include <string.h>
#include <math.h>
using namespace std;

int N;
double dist[20][20], memo[1 << 16]; // 1 << 16 is 2^16, recall that max N = 8

double matching(int bit_mask) {
  if (memo[bit_mask] > -0.5) // see that we initialize the array with -1 in the main function
    return memo[bit_mask];
  if (bit_mask == (1 << 2 * N) - 1) // all are matched
    return memo[bit_mask] = 0;

  double matching_value = 32767 * 32767; // initialize with large value
  for (int p1 = 0; p1 < 2 * N; p1++)
    if (!(bit_mask & (1 << p1))) { // if this bit is off
      for (int p2 = p1 + 1; p2 < 2 * N; p2++)
        if (!(bit_mask & (1 << p2))) // if this different bit is also off
          matching_value = min(matching_value,
            dist[p1][p2] + matching(bit_mask | (1 << p1) | (1 << p2)));
      break; // this 'break' is necessary. do you understand why?
    }        // hint: it helps reducing time complexity from O((2N)^2 * 2^(2N)) to O((2N) * 2^(2N))

  return memo[bit_mask] = matching_value;
}

int main() {
  char line[1000], name[1000];
  int i, j, caseNo = 1, x[20], y[20];
  // freopen("10911.txt", "r", stdin); // one way to simplify testing
  while (sscanf(gets(line), "%d", &N), N) {
    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++)
        dist[i][j] = sqrt((double)(x[i] - x[j]) * (x[i] - x[j]) + (y[i] - y[j]) * (y[i] - y[j]));

    // using DP to solve matching on general graph
    memset(memo, -1, sizeof memo);
    printf("Case %d: %.2lf\n", caseNo++, matching(0));
  }
  return 0;
}
```

1.2.1 Tip 1: Quickly Identify Problem Types

In ICPCs, the contestants will be given a **set** of problems (≈ 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. For IOI, please refer to IOI syllabus 2009 [8] and [28].

No	Category	Sub-Category	In This Book	Appearance Frequency
1.	Ad Hoc	Straightforward	Section 1.3	1-2
		Simulation	Section 1.3	0-1
2.	Complete Search	Iterative	Section 3.1	0-1
		Backtracking	Section 3.1	0-1
3.	Divide & Conquer		Section 3.2	0-1
4.	Greedy	Classic	Section 3.3.1	≈0
		Original	Section 3.3.2	1
5.	Dynamic Programming	Classic	Section 3.4.2	≈0
		Original	Section 3.4.3	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			0-1
			Total in Set	7-16 (usually ≤ 11)

Table 1.1: Recent ACM ICPC Asia Regional Problem Types

The classification in Table 1.1 is adapted from [18] and by no means complete. Some problems, e.g. 'sorting', are not classified here as they are 'trivial' and only used as 'sub-routine' in a bigger problem. We do not include 'recursion' as it is embedded in other categories. We also omit 'data structure related problems' and such problems will be categorized as 'Ad Hoc'.

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 is either a solution for graph problem: All-Pairs Shortest Paths (APSP, Section 4.7) or a Dynamic Programming (DP) algorithm (Section 3.4).

In the future, these classifications may grow or 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 recent ICPC World Finals 2010.

As an exercise, read the UVa [17] problems shown in Table 1.2 and determine their problem types. The first one has been filled for you. Filling this table is easy after mastering this book.

UVa	Title	Problem Type	Hint
10360	Rat Attack	Complete Search or Dynamic Programming	Section 3.1 or 3.4
10341	Solve It		Section 3.2
11292	Dragon of Loowater		Section 3.3
11450	Wedding Shopping		Section 3.4
11635	Hotel Booking		Section 4.3 + 4.5
11506	Angry Programmer		Section 4.8
10243	Fire! Fire!! Fire!!!		Section 4.9.1
10717	Mint		Section 5.3.2
11512	GATTACA		Section 6.4
10065	Useless Tile Packers		Section 7.3

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	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 or may not be able to solve it now

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 2: Do Algorithm Analysis

Once you have designed an algorithm to solve a particular problem in a programming contest, you must now 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 and some of them are not fast enough... 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 correct answer)!

For example, the maximum size of input n is $100K$, or 10^5 ($1K = 1,000$), and your algorithm is of order $O(n^2)$. Your common sense told 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 order $1M$, or 10^6 ($1M = 1,000,000$) operations in seconds, your common sense told you that this one likely able to pass the time limit.

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 relatively small computation time.

So, by analyzing the complexity of your algorithm with the given input bound and stated time/memory limit, you can do a better judging whether you should try coding your algorithm (which will take your time, especially in the time-constrained ICPCs and IOIs), or 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. We assume that you have this basic skill. Please check this reference book: "Introduction to Algorithms" [4] and make sure you understand how to:

- Prove correctness of an algorithm (especially for Greedy algorithms, see Section 3.3).
- Analyze time/space complexity analysis for iterative and recursive algorithms.
- Perform amortized analysis (see [4], Chapter 17) – although rarely used in contests.
- Do output-sensitive analysis, to analyze algorithm which depends on output size, example: the $O(|Q| + occ)$ complexity for finding an exact string matching of query string Q with help of Suffix Tree (see Section 6.4).

Many novice programmers usually skip this phase and tempted to directly code the first algorithm that they can think of (usually the naïve version), after that they ended up realizing that the chosen data structure is not efficient or their algorithm is not fast enough (or wrong). **Our advice: 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 programming contest perspective. Usually, the input size constraints are given in the problem description. Using some logical assumptions that typical year 2010 CPU can do $1M$ operations in 1s and time limit of 3s (typical time limit used in most UVa online judge [17] 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 already pass the time limit, just use it!

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 3.1.2 later, we will see a few tips that may allow us to enlarge the possible input size n for the same class of algorithm.

n	Worst AC Algorithm	Comment
≤ 10	$O(n!), O(n^6)$	e.g. Enumerating a Permutation
≤ 20	$O(2^n), O(n^5)$	e.g. DP + Bitmask Technique
≤ 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
$\leq 1K$	$O(n^2)$	e.g. Bubble/Selection/Insertion Sort
$\leq 100K$	$O(n \log_2 n)$	e.g. Merge Sort, building Segment Tree
$\leq 1M$	$O(n), O(\log_2 n), O(1)$	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 2010 CPU can compute $1M$ items in 1s and Time Limit of 3s.

Additionally, we have a few other rules of thumb:

- $2^{10} \approx 10^3$, $2^{20} \approx 10^6$.
- Max 32-bit signed integer: $2^{31} - 1 \approx 2 \times 10^9$ (or up to ≈ 9 decimal digits);
  Max 64-bit signed integer (long long) is $2^{63} - 1$: 9×10^{18} (or up to ≈ 18 decimal digits).
  Use 'unsigned' if slightly higher positive number is needed $[0 \ldots 2^{64} - 1]$.
  If you need to store integers $\geq 2^{64}$, you need to use the Big Integer technique (Section 5.4).
- 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.
- Dynamic Programming algorithms which fill a 2-D matrix in $O(k)$ per cell is in $O(k \times n^2)$.
- 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.

As an exercise for this section, please answer the following questions:

1. There are n webpages ($1 \leq n \leq 10M$). Each webpage i has different page rank r_i. You want to pick 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 top 10.
   (b) Use priority queue data structure (heap) (Section 2.2.2).
2. Given a list L of up to $10K$ integers, you want 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) Balanced Binary Search Tree (Section 2.2.2).
   (c) Hash Table (Section 2.2.2).
   (d) Segment Tree (Section 2.3.3).
   (e) Suffix Tree (Section 6.4).
   (f) Simple Array that is pre-processed with Dynamic Programming (Section 2.2.1 & 3.4).
3. You have to compute the 'shortest path' between two vertices on a weighted Directed Acyclic Graph (DAG) with $|V|, |E| \leq 100$K. Which algorithm(s) can be used?
   (a) Dynamic Programming + Topological Sort (Section 3.4, 4.2, & 4.9.2).
   (b) Breadth First Search (Section 4.3).
   (c) Dijkstra's (Section 4.5).
   (d) Bellman Ford's (Section 4.6).
   (e) Floyd Warshall's (Section 4.7).
4. Which algorithm is faster (based on its time complexity) for producing a list of the first $10K$ prime numbers? (Section 5.3.1)
   (a) Sieve of Eratosthenes (Section 5.3.1).
   (b) For each number $i \in [1 - 10K]$, test if i is a prime with prime testing function.

1.2.3 Tip 3: Master Programming Languages

There are several programming languages allowed in ICPC, including C/C++ and Java. Which one should we master? Our experience gives us the following answer: although we prefer C++ with built-in Standard Template Library (STL), we still need to master Java, albeit slower, since this language has a powerful BigInteger, String Processing, and GregorianCalendar API. Simple illustration is shown below (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 data structure (`unsigned long long`: $2^{64} - 1$) in C/C++. Using C/C++, you will hard time coding this simple problem as there is no native support for Big Integer data structure in C/C++ yet. Meanwhile, the Java code is simply this:

```
import java.util.*;
import java.math.*;

class Main { // standard class name in UVa OJ
  public static void main(String[] args) {
    BigInteger fac = new BigInteger.valueOf(1); // :)
    for (int i = 2; i <= 25; i++)
      fac = fac.multiply(BigInteger.valueOf(i)); // wow :)
    System.out.println(fac);
} }
```

Another illustration to reassure you that mastering a programming language is good: Read this input: There are N lines, each line always start with character '0' followed by '.', then unknown number of digits x, finally each line always terminated with three dots "...". See an example below.

```
2
0.1227...
0.517611738...
```

One solution is as follows:

```
#include <iostream> // or <cstdio>
using namespace std;
char digits[100]; // using global variables in contests can be a good strategy

int main() {
  scanf("%d", &N);
  while (N--) { // we simply loop from N, N-1, N-2, ... 0
    scanf("0.%[0-9]...", &digits); // surprised?
    printf("the digits are 0.%s\n", digits);
} }
```

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 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 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 bug-free code and you can do it fast! 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.

1. Given a string that represents a base X number, e.g. FF (base 16, Hexadecimal), convert it to base Y, e.g. 255 (base 10, Decimal), $2 \leq X, Y \leq 36$. (More details in Section 5.4.2).
2. Given a list of integers L of size up to $1M$ items, determine whether a value v exists in L? (More details in Section 2.2.1).
3. Given a date, determine what is the day (Monday, Tuesday, ..., Sunday) of that date?
4. Given a long string, replace all the occurrences of a character followed by two consecutive digits in with "***", e.g. S = "a70 and z72 will be replaced, but aa24 and a872 will not" will be transformed to S = "*** and *** will be replaced, but aa24 and a872 will not".

1.2.4 Tip 4: Master the Art of Testing Code

You thought 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 solutoin 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 (2009 rule), there exists a partial credit system, in which you will get scored based on the number of correct/total number of test cases for the latest code that you have submitted for that problem, but the judging will only be done *after* the contest is over, so you must be very sure that your code is doing OK.

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 your code's correctness.

Rather than wasting submissions (and get time or point penalties) by getting non AC responses, you may want to design some tricky test cases first, test it in your own machine, and ensure 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 :). This concept is also used in TopCoder [26] 'challenge phase'.

Here are some guidelines for designing good test cases, based on our experience:

1. Must include sample input as you have the answer given... Use 'fc' in Windows or 'diff' in UNIX to help checking your code's output against the sample output.
2. Must include boundary cases. Increase the size of input incrementally up to the maximum possible. Sometimes your program works for small input size, but behave wrongly when input size increases. Check for overflow, out of bounds, etc.
3. For multiple input test cases, use two identical test cases consecutively. Both must output the same result. This is to check whether you have forgotten to initialize some variables, which will be easily identified if the 1st instance produce correct output but the 2nd one does not.
4. Create tricky test cases by identifying cases that are 'hidden' in the problem description.
5. Do not assume the input will always be nicely formatted if the problem description does not say so (especially for badly written programming problem). Try inserting white spaces (space, tabs) in your input, and check whether your code is able to read in the values correctly.
6. Finally, generate large random test cases. See if your code terminates on time and still give reasonably ok output (correctness is hard to verify here – this test is only to verify that your code runs within time limit).

However, after all these careful steps, you may still get non-AC responses. In ICPC, 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:

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 the 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 the 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?

1.2.5 Tip 5: 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 websites that can help you improve your problem solving skill. Success is a continuous journey!

University of Valladolid (from Spain) Online Judge [17] contains past years ACM contest problems (usually local or regional) plus problems from another 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 (9 August 2010), Steven is ranked 121 (for solving 857 problems) while Felix is ranked 70 (for solving 1089 problems) from ≈ 100386 UVa users and 2718 problems.

Figure 1.1: University of Valladolid (UVa) Online Judge, a.k.a Spanish OJ [17]

UVa 'sister' online judge is the ACM ICPC Live Archive that contains 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: ACM ICPC Live Archive [11]

USA Computing Olympiad has a very useful training website [18] for you to learn about programming contest. This one is more geared towards IOI participants. Go straight to their website, register your account, and train yourself.

Figure 1.3: USACO Training Gateway [18]

TopCoder arranges frequent 'Single Round Match' (SRM) [26] that consists of a few problems that should be solved in 1-2 hours. Then 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 a more diligent contestants who happen to solve 'more' easier problems.

Figure 1.4: TopCoder [26]

1.3 Getting Started: Ad Hoc Problems

We will end this chapter by introducing you to the first problem type in ICPC: the Ad Hoc problems. According to USACO training gateway [18], Ad Hoc problems are problems that 'cannot be classified anywhere else', where each problem description and the corresponding solution are 'unique'. Ad Hoc problems can be further classified into two: straightforward – where the solution just requires translation of problem requirement to code; or simulation problem – where there are some set of rules that must be simulated to obtain the answer.

Ad Hoc problems almost usually appear in a programming contest. Using a benchmark of total 10 problems, there may be 1-2 Ad Hoc problems. If the Ad Hoc problem is easy, it will usually be the first problem being attacked by 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 lower half (rank 30-60) if your team can *only* do this type of problem during an ICPC regional contest.

Get your coding skills up and running by solving these Ad Hoc problems before continuing to the next chapter. We have selected one Ad Hoc problem from every volume in UVa online judge [17] (there are 28 volumes as of 9 August 2010) plus several ones from ACM ICPC Live Archive [11]. Note that some 'simple' Ad Hoc problems below are 'tricky'.

Programming Exercises related to Ad Hoc problems:

1. UVa 100 - The 3n + 1 problem (follow the problem description, note the term 'between'!)
2. UVa 272 - TEX Quotes (simply replace all double quotes to TEX() style quotes)
3. UVa 394 - Mapmaker (array manipulation)
4. UVa 483 - Word Scramble (read char by char from left to right)
5. UVa 573 - The Snail (be careful of boundary cases!)
6. UVa 661 - Blowing Fuses (simulation)
7. UVa 739 - Soundex Indexing (straightforward conversion problem)
8. UVa 837 - Light and Transparencies (sort the x-axis first)
9. UVa 941 - Permutations (find the n-th permutation of a string, simple formula exists)
10. UVa 10082 - WERTYU (keyboard simulation)
11. UVa 10141 - Request for Proposal (this problem can be solved with one linear scan)
12. UVa 10281 - Average Speed (distance = speed × time elapsed)
13. UVa 10363 - Tic Tac Toe (simulate the Tic Tac Toe game)
14. UVa 10420 - List of Conquests (simple frequency counting)
15. UVa 10528 - Major Scales (the music knowledge is given in the problem description)
16. UVa 10683 - The decadary watch (simple clock system conversion)
17. UVa 10703 - Free spots (array size is 'small', 500 x 500)
18. UVa 10812 - Beat the Spread (be careful with boundary cases!)
19. UVa 10921 - Find the Telephone (simple conversion problem)
20. UVa 11044 - Searching for Nessy (one liner code exists)
21. UVa 11150 - Cola (be careful with boundary cases!)
22. UVa 11223 - O: dah, dah, dah! (tedious morse code conversion problem)
23. UVa 11340 - Newspaper (use 'Direct Addressing Table' to map char to integer value)
24. UVa 11498 - Division of Nlogonia (straightforward problem)
25. UVa 11547 - Automatic Answer (one liner code exists)
26. UVa 11616 - Roman Numerals (roman numeral conversion problem)
27. UVa 11727 - Cost Cutting (sort the 3 numbers and get the median)
28. UVa 11800 - Determine the Shape (Ad Hoc geometry problem)
29. LA 2189 - Mobile Casanova (Dhaka06)
30. LA 3012 - All Integer Average (Dhaka04)
31. LA 3173 - Wordfish (Manila06) (STL `next_permutation`, `prev_permutation`)
32. LA 3996 - Digit Counting (Danang07)
33. LA 4202 - Schedule of a Married Man (Dhaka08)
34. LA 4786 - Barcodes (World Finals Harbin10)

1.4 Chapter Notes

Figure 1.5: Some Reference Books that Inspired the Authors to Write This Book

This and subsequent chapters are supported by many text books (see Figure 1.5) and Internet resources. Tip 1 is an adaptation from introduction text in USACO training gateway [18]. More details about Tip 2 can be found in many CS books, e.g. Chapter 1-5, 17 of [4]. Reference for Tip 3 are `http://www.cppreference.com`, `http://www.sgi.com/tech/stl/` for C++ STL and `http://java.sun.com/javase/6/docs/api` for Java API. For more insights to do better testing (Tip 4), a little detour to software engineering books may be worth trying. There are many other Online Judges than those mentioned in Tip 5, e.g.
SPOJ `http://www.spoj.pl`,
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.

There are approximately **34 programming exercises** discussed in this chapter.

Chapter 2

Data Structures and Libraries

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

This chapter acts as a foundation for subsequent chapters.

2.1 Data Structures

Data structure is 'a way to store and organize data' in order to support efficient insertions, queries, searches, updates, and deletions. Although a data structure in 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 a difference between passing or exceeding the problem's time limit. There are many ways to organize the same data and sometimes one way is better than the other on different context, as we will see in the discussion below. Familiarity with the data structures discussed in this chapter is a must in order to understand the algorithms 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 will **not** review them again in this book. We simply highlight the fact that they all have built-in libraries in C++ STL and Java API (Note that in this version of the book, we write *most* example codes from C++ perspective). If you feel that you are not sure with any of the terms or data structures mentioned in Section 2.2, pause reading this book, quickly explore and learn that term in the reference books, e.g. [3][1], and resume when you get the *basic ideas* of those data structures.

Note that for competitive programming, you just have to be able to *use* (i.e. know the strengths, weaknesses, and time/space complexities) a certain data structure to solve the appropriate contest problem. Its theoretical background is good to know, but can be skipped.

This chapter is divided into two parts. Section 2.2 contains basic data structures with their basic operations that currently have built-in libraries. 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.

[1]Materials in Section 2.2 are usually taught in level-1 'data structures and algorithms' course in CS curriculum. High school students who are planning to join competitions like 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 in C/C++ and in 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 normally mentioned in a programming problem, then usually the declared array size is this value + small extra buffer. Typical dimensions of the array are: 1-D, 2-D, 3-D, and rarely goes beyond 4-D. Typical operations for array are: accessing certain indices, sorting the array, linearly scanning, or binary searching the array.
- Resizeable Array a.k.a. Vector: C++ STL <`vector`> (Java `ArrayList`)
  All else the same as static array but has auto-resize feature. Using `vector` over array is better if array size is unknown beforehand, i.e. before running the program. Usually, we initialize the size with some guess value for better performance. Typical operations are: `push_back()`, `at()`, `[]` operator, `erase()`, and typically use iterator to scan the content of the `vector`.

Efficient Sorting and Searching in Static/Resize-able Array

There are two central operations commonly performed on array: sorting and searching.

There are *many* sorting algorithms mentioned in CS textbooks, which we classify as:

1. $O(n^2)$ comparison-based sorting algorithms [4]: 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 [4]: Merge/Heap/Random Quick Sort.
   We can use C++ STL `sort`, `partial_sort`, `stable_sort`, in <`algorithm`> to achieve this purpose (Java `Collections.sort`). We only need to specify the required comparison function and these library routines will handle the rest.
3. Special purpose sorting algorithms [4]: $O(n)$ Counting Sort, Radix Sort, 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.

Then, 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> (or 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).

- 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 theoretically slow for accessing data as it has to be performed from the head or tail of a list.

- Stack: C++ STL <stack> (Java Stack)
  This data structure is used as part of algorithm to solve a certain problem (e.g. Postfix calculation, Graham's scan in Section 7.3). Stack only allows insertion (push) and deletion (pop) from the top only. This behavior is called Last In First Out (LIFO) as with normal stack 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)
  This data structure is used in algorithms like Breadth First Search (BFS) (Section 4.3). A queue only allows insertion (enqueue) from the back (rear), and only allows deletion (dequeue) from the head (front). This behavior is called First In First Out (FIFO), similar to normal queue in the real world. Typical operations are push()/pop() (insert from back/take out from front of queue), front()/back() (obtain content from the front/back of queue), empty().

2.2.2 Non-Linear Data Structures

For some computational problems, there are better ways to organize data other than ordering it sequentially. With efficient implementation of non-linear data structures shown below, you can search items much faster, 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 requires $O(n)$ 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 a 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 (or equal) than x. 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 those actions (details in [4]) – but this only works if the BST is balanced.

  Implementing a *bug-free* balanced BST like AVL[2] Tree or Red-Black (RB) Tree is tedious and hard to do under time constrained contest environment. Fortunately, C++ STL has <map> and <set> which are usually the implementation of RB Tree, thus all operations are in $O(\log n)$. Mastery of these two STL templates can save a lot of precious coding time during contests! The difference is simple: <map> stores (key → data) pair whereas <set> only stores the key.

- Heap: C++ STL <queue>: 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*. Instead of enforcing BST property, Heap enforces Heap property: In each subtree rooted at x, items on the left and the right subtrees of x are smaller than x (see Figure 2.1, right). This property guarantees that the top of the heap is the maximum element. There is usually no notion of 'search' in Heap, but only insertion and deletion, which can be easily done by traversing a $O(\log n)$ leaf-to-root or root-to-leaf path [4].

[2]However, for some tricky contest problems where you have to 'augment your data structure' (see Chapter 14 of [4]), AVL Tree knowledge is needed as C++ STL <map> or <set> cannot be augmented easily.

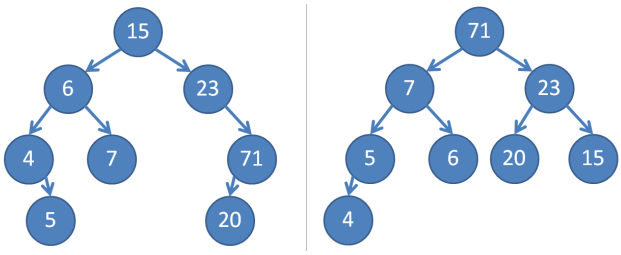


Figure 2.1: Examples of BST (Left) and Heap (Right)

Heap is useful to model Priority Queue, where item with highest priority can be deleted in $O(\log n)$ and new item can be inserted into priority queue also in $O(\log n)$. The implementation of `priority_queue` is available in C++ STL <queue>. Priority Queue is an important component in algorithms like Kruskal's for Minimum Spanning Tree (MST) problem (Section 4.4) and Dijkstra's for Single-Source Shortest Paths (SSSP) problem (Section 4.5).

This data structure is also used to perform `partial_sort` in C++ STL <algorithm>. This is done by taking the max element k times (k is the number of the top most items to be sorted). As each delete-max is in $O(\log n)$, `partial_sort` has $O(k \log n)$ time complexity.

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

  However, a form of Hash Table is actually used in contests, namely 'Direct Addressing Table' (DAT), where the key itself determines the index, bypassing the need of 'hash function'. For example, UVa 11340 - Newspaper, where all possible ASCII characters [0-255] are assigned to certain monetary values, e.g. 'a' → '3', 'W' → '10', ..., 'I' → '13'.

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

- Static array, C++ STL <vector>, <bitset>, Direct Addressing Table
  1. UVa 482 - Permutation Arrays (simple array manipulation)
  2. UVa 594 - One Two Three Little Endian (manipulate bit string easily with <bitset>)
  3. UVa 11340 - Newspaper (Direct Addressing Table)
- C++ STL <algorithm>
  1. UVa 146 - ID Codes (use `next_permutation`)
  2. UVa 10194 - Football a.k.a. Soccer (multi-fields sorting, use `sort`)
  3. UVa 10258 - Contest Scoreboard (multi-fields sorting, use `sort`)
- Sorting-related problems
  1. UVa 299 - Train Swapping (inversion index[3] problem solvable with bubble sort)
  2. UVa 612 - DNA Sorting (inversion index + `stable_sort`)
  3. UVa 10810 - Ultra Quicksort (inversion index - requires $O(n \log n)$ merge sort)

[3]Inversion index problem: count how many swaps are needed to make the list sorted.

4. UVa 11462 - Age Sort (counting sort problem, see [4])
5. UVa 11495 - Bubbles and Buckets (inversion index - requires $O(n \log n)$ merge sort)

- C++ STL <stack>
  1. UVa 127 - "Accordian" Patience (shuffling <stack>)
  2. UVa 514 - Rails (use <stack> to simulate the process)
  3. UVa 673 - Parentheses Balance (classical problem)
  4. UVa 727 - Equation (Infix to Postfix conversion)
- C++ STL <queue>
  1. UVa 336 - A Node Too Far (<queue> used inside BFS, Section 4.3)
  2. UVa 10901 - Ferry Loading III (simulation with <queue>)
  3. UVa 11034 - Ferry Loading IV (simulation with <queue>)
- C++ STL <map>/<set>
  1. UVa 10226 - Hardwood Species (use <map>)
  2. UVa 11239 - Open Source (use <map> and <set> to check previous strings efficiently)
  3. UVa 11308 - Bankrupt Baker (use <map> and <set> to help managing the data)
  4. UVa 11136 - Hoax or what (use multiset in <set>)
- C++ STL priority_queue in <queue>
  1. UVa 908 - Re-connecting Computer Sites (priority_queue in Kruskal's, Section 4.4)
  2. UVa 11492 - Babel (priority_queue in Dijkstra's, Section 4.5)
  3. LA 3135 - Argus (Beijing04)

2.3 Data Structures with Our-Own Libraries

As of 9 August 2010, 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 a bug-free implementations during contests. In this section, we only discuss the ideas of these data structures.

2.3.1 Graph

Graph is a pervasive data structure which appears in many CS problems. Graph is simply a collection of vertices and edges (that store connectivity information between those vertices). In Chapter 3 & 4, we will explore many important graph problems and algorithms. In this subsection, we only briefly discuss four basic ways (there are others) to store graph information. Assuming that we have a graph G with V vertices and E edges, here are the ways to store them:

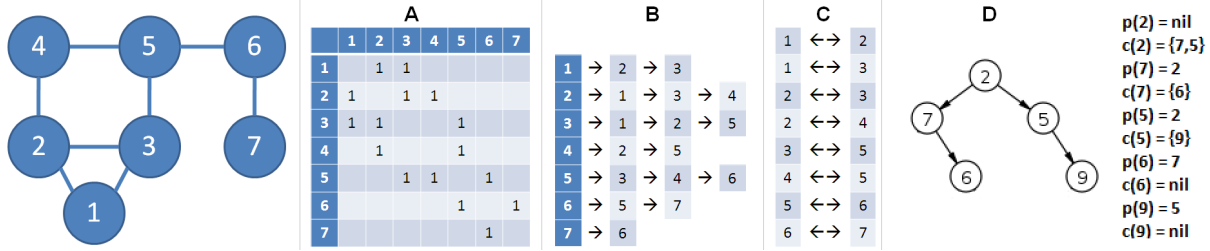


Figure 2.2: Example of various Graph representations

A Adjacency Matrix, usually in form of 2-D array.

In contest problems involving graph, usually V is known, thus we can build a 'connectivity table' by setting up a 2-D, $O(V^2)$ static array: `int AdjMat[V][V]`. 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 asked, but it is not good for *large sparse graph* as there will be too many cells in the 2-D array that are blank (contain zeroes). An adjacency Matrix requires exactly $O(V)$ 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 Adjacency List.

B Adjacency List, usually in form of C++ STL `vector<vii> AdjList`, with `vii` defined as: `typedef pair<int, int> ii; typedef vector<ii> vii; // our data type shortcuts`

In Adjacency List, we have a `vector` of V vertices and for each vertex v, we store another `vector` that contains **pairs** of (neighboring vertex and it's edge weight) that have connection to v. If the graph is unweighted, simply store weight = 0 or drop this second attribute.

With Adjacency List, we can enumerate the list of neighbors of a vertex v efficiently. If there are k neighbors of v, this enumeration is $O(k)$. As this is one of the most common operations in most graph algorithms, it is advisable to stick with Adjacency List as your default choice.

C Edge List, usually in form of C++ STL `priority_queue<pair<int, ii> > EdgeList`.

In Edge List, we store the list of edges, usually in some order. This structure is very useful for Kruskal's algorithm for MST (Section 4.4) where the collection of edges are sorted by their length from shortest to longest.

D Parent-Child Tree Structure, usually in form of `int parent` & C++ STL `vector<int> child`.

If the graph is a tree (connected graph with no cycle and $E = V - 1$), like a directory/folder structure, then there exists another form of data structure. For each vertex, we only store two attributes: the parent (`NULL` for root vertex) and the list of children (`NULL` for leaves).

Exercise: Show the Adjacency Matrix, Adjacency List, and Edge List of the graph in Figure 4.1.

2.3.2 Union-Find Disjoint Sets

Union-Find Disjoint Sets is a data structure to model a collection of disjoint sets which has abilities to efficiently[4] 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. These two operations are useful for algorithms like Kruskal's (Section 4.4) or problems that involve 'partitioning', like keeping track of connected components of an undirected graph (see Section 4.2).

These seemingly simple operations are not *efficiently* supported by C++ STL <set> which only deals with a single set. Having a `vector` of `set`s and looping through each one to find which set

[4] M operations of this data structure runs in $O(M \times \alpha(n))$, but $\alpha(n)$ is just less than 5 for most practical values of n. For more details, see [33].

an item belongs to is expensive! C++ STL <algorithm>'s `set_union` 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`! Thus, we need our own library to support this data structure. One such example is shown in this section.

The key ideas of this data structure are like this: Keep a representative ('parent') item of each set. This information is stored in `vector<int> pset`, where `pset[i]` tells the representative item of the set that contains item `i`. Example: suppose we have 5 items: {A, B, C, D, E} as 5 disjoint sets of 1 item each. Each item initially has itself as the representative, as shown in Figure 2.3.

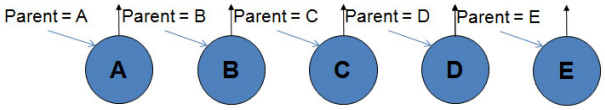


Figure 2.3: Calling `initSet()` to Create 5 Disjoint Sets

When we want to merge two sets, we call `unionSet(i, j)` which make both items 'i' and 'j' to have the same representative item[5] – directly or indirectly (see Path Compression below). This is done by calling `findSet(j)` – what is the representative of item 'j', and assign that value to `pset[findSet(i)]` – update the parent of the representative item of item 'i'.

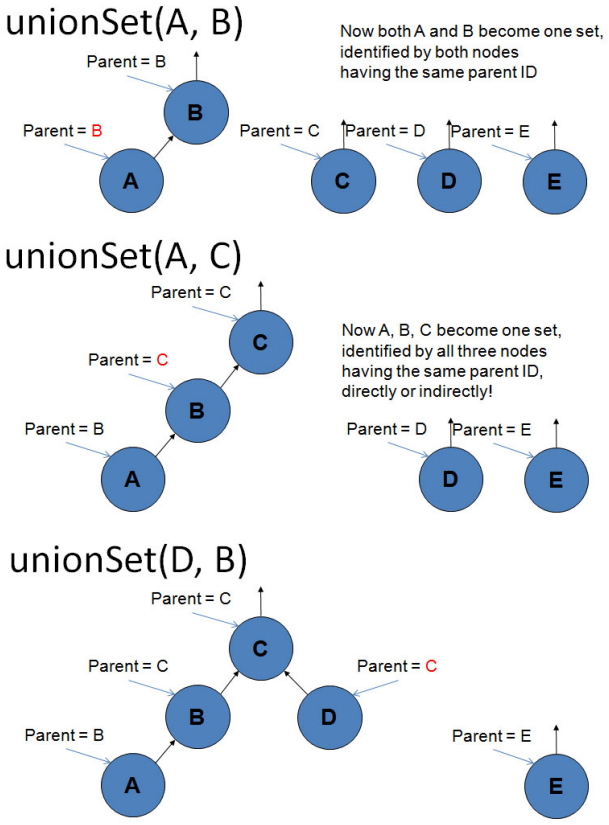


Figure 2.4: Calling `unionSet(i, j)` to Union Disjoint Sets

In Figure 2.4, we see what is happening when we call `unionSet(i, j)`: every union is simply done by changing the representative item of one item to point to the other's representative item.

[5]There is another heuristic called 'union-by-rank' [4] that can further improve the performance of this data structure. But we omit this enhancing heuristic from this book to simplify this discussion.

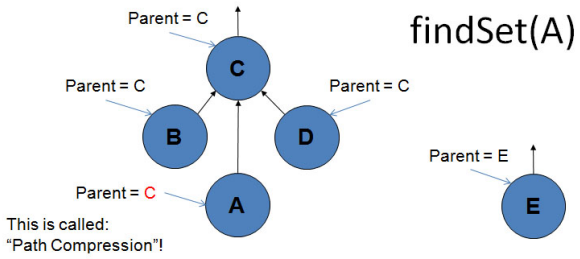


Figure 2.5: Calling `findSet(i)` to Determine the Representative Item (and Compressing the Path)

In Figure 2.5, we see what is happening when we call `findSet(i)`. This function 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 heuristic strategy is aptly named as 'Path Compression'.

In Figure 2.6, we illustrate another operation for this data structure, called `isSameSet(i, j)` that 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.

Our library implementation for Union-Find Disjoint Sets is shown below.

```
#define REP(i, a, b) \ // all codes involving REP uses this macro
  for (int i = int(a); i <= int(b); i++)

vector<int> pset(1000); // 1000 is just an initial number, it is user-adjustable.
void initSet(int _size) { pset.resize(_size); REP (i, 0, _size - 1) pset[i] = i; }
int findSet(int i) { return (pset[i] == i) ? i : (pset[i] = findSet(pset[i])); }
void unionSet(int i, int j) { pset[findSet(i)] = findSet(j); }
bool isSameSet(int i, int j) { return findSet(i) == findSet(j); }
```

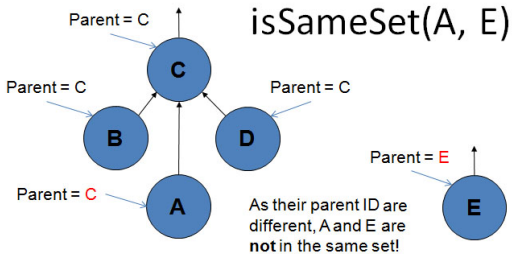


Figure 2.6: Calling isSameSet(i, j) to Determine if Both Items Belong to the Same Set

Exercise 1: There are two more queries commonly performed on the Union-Find Disjoint Sets data structure: `int numberOfSets()` 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: In [4], there is a 'union by rank' heuristic to speed-up this data structure. Do you think this heuristic will help speed up the data structure significantly? If yes, in which case(s)? Is there any programming tricks to achieve similar effect without using this heuristic?

2.3.3 Segment Tree

In this subsection, we discuss another data structure which can efficiently answer *dynamic* range queries. As a starting point, we discuss a problem of finding the index of the minimum element in an array given a range: $[i..j]$. This is more commonly known as the Range Minimum Query (RMQ). For example, given an array A of size 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 array A below, RMQ(3, 4) = 4, RMQ(0, 0) = 0, RMQ(0, 1) = 1, and RMQ(0, 6) = 5.

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

There are several ways to solve this RMQ. One of the 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 algorithm maybe infeasible.

In this section, we solve the RMQ with Segment Tree: a binary tree similar to heap, but usually not a complete binary tree. For the array A above, the segment tree is shown in Figure 2.7. The root of this tree contains the full segment, from [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_segment_tree` routine below. With segment tree ready, answering an RMQ can now be done in $O(\log n)$.

For example, we want to answer RMQ(1, 3). The execution in Figure 2.7 (red solid lines) is as follows: From root [0, 6], we know that the answer for RMQ(1, 3) is on the left of vertex [0, 6] as [0, 6] is still larger than the RMQ(1, 3), thus the stored min(imum) value of [0, 6] = 5 is not appropriate as it is the min value over a larger segment [0, 6] than the RMQ(1, 3).

We move to the left segment [0, 3]. At vertex [0, 3], we have to search two sides as [0, 3] is still larger than the RMQ(1, 3) and intersect *both* the left segment [0, 1] and the right segment [2, 3].

The right segment is [2, 3], which is inside the required RMQ(1, 3), so from the stored min value inside this node, we know that RMQ(2,3) = 2. We do *not* need to traverse further down.

The left segment is [0, 1], which is not yet inside the RMQ(1, 3), so another split is necessary. From [0, 1], we move right to segment [1, 1], which is now inside the RMQ(1, 3). Then, we return the min value = 1 to the caller.

Back in segment [0, 3], we now know that RMQ(1, 1) = 1 and RMQ(2, 3) = 2. Because A[RMQ(1, 1)] > A[RMQ(2, 3)] since A[1] = 7 and A[2] = 3, we know that RMQ(1, 3) = 2.

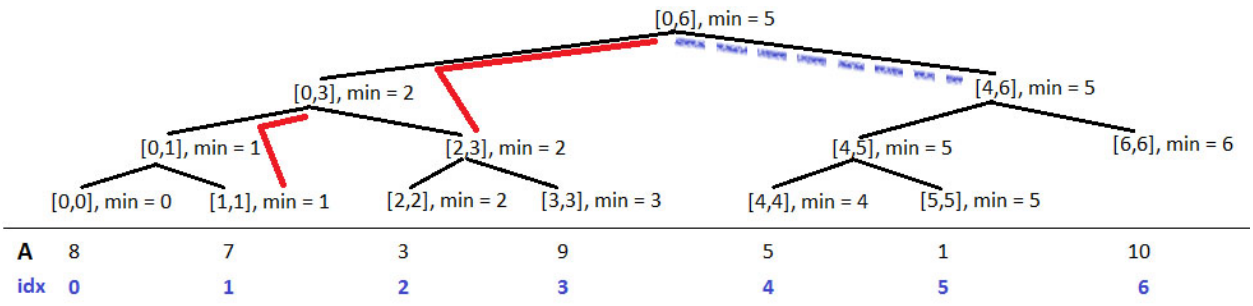


Figure 2.7: Segment Tree of Array A {8, 7, 3, 9, 5, 1, 10}

Let's take a look at another example: RMQ(4, 6). The execution in Figure 2.7 (blue dashed line) is as follows: We again start from the root [0, 6]. Since it is bigger than the query, 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 node, 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 $O(\log n)$. 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 an 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 3.4.3.

The Segment Tree becomes useful if array A is frequently updated. For example, if A[5] is now changed from 1 to 100, then what we need to do is to update the leaf to root nodes which can be done in $O(\log n)$. The DP solution requires another $O(n \log n)$ pre-processing to do the same.

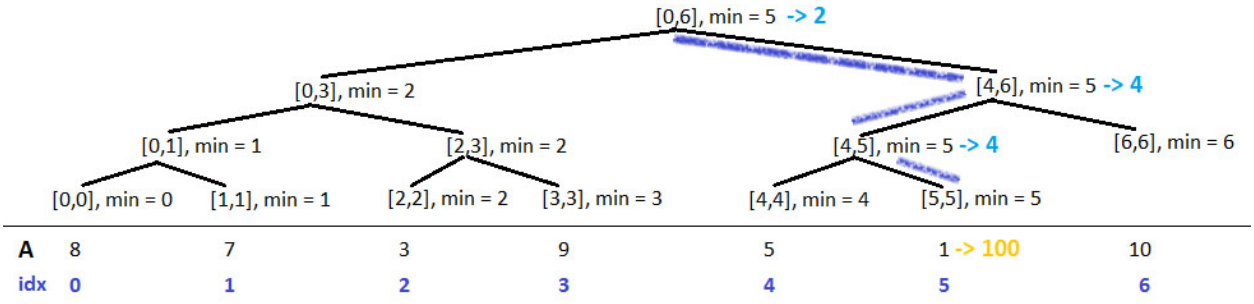


Figure 2.8: Updating Array A to {8, 7, 3, 9, 5, **100**, 10}. Only leaf-to-root nodes are affected.

Our library implementation for Segment Tree is shown below. The code shown here supports *static* Range Minimum/Maximum/Sum queries (the dynamic update part is left as exercise). There are of course other ways to implement segment tree, e.g. a more efficient version that only expands the segments when needed.

```
#include <iostream>
#include <math.h>
#include <vector>
using namespace std;

// Segment Tree Library
// The segment tree is stored like a heap array
#define RANGE_SUM 0
#define RANGE_MIN 1
#define RANGE_MAX 2
vi segment_tree;

void init_segment_tree(int N) { // if original array size is N,
  // the required segment_tree array length is 2*2^(floor(log2(N)) + 1);
  int length = (int)(2 * pow(2.0, floor((log((double)N) / log(2.0)) + 1)));
  segment_tree.resize(length, 0); // resize this vector and fill with 0
}

void build_segment_tree(int code, int A[], int node, int b, int e) {
  if (b == e) { // as b == e, either one is fine
    if (code == RANGE_SUM) segment_tree[node] = A[b]; // store value of this cell
    else                   segment_tree[node] = b; // if RANGE_MIN/MAXIMUM, store index
  }
```

```
  else { // recursively compute the values in the left and right subtrees
    int leftIdx = 2 * node, rightIdx = 2 * node + 1;
    build_segment_tree(code, A, leftIdx , b              , (b + e) / 2);
    build_segment_tree(code, A, rightIdx, (b + e) / 2 + 1, e          );
    int lContent = segment_tree[leftIdx], rContent = segment_tree[rightIdx];

    if (code == RANGE_SUM) // make this segment contains sum of left and right subtree
      segment_tree[node] = lContent + rContent;
    else { // (code == RANGE_MIN/MAXIMUM)
      int lValue = A[lContent], rValue = A[rContent];
      if (code == RANGE_MIN) segment_tree[node] = (lValue <= rValue) ? lContent : rContent;
      else                   segment_tree[node] = (lValue >= rValue) ? lContent : rContent;
} } }

int query(int code, int A[], int node, int b, int e, int i, int j) {
  if (i > e || j < b) return -1; // if the current interval does not intersect query interval
  if (b >= i && e <= j) return segment_tree[node]; // if the current interval is inside query interval

  // compute the minimum position in the left and right part of the interval
  int p1 = query(code, A, 2 * node    , b              , (b + e) / 2, i, j);
  int p2 = query(code, A, 2 * node + 1, (b + e) / 2 + 1, e          , i, j);

  // return the position where the overall minimum is
  if (p1 == -1) return p2; // can happen if we try to access segment outside query
  if (p2 == -1) return p1; // same as above

  if (code == RANGE_SUM)      return p1 + p2;
  else if (code == RANGE_MIN) return (A[p1] <= A[p2]) ? p1 : p2;
  else                        return (A[p1] >= A[p2]) ? p1 : p2;
}

int main() {
  int A[] = {8,7,3,9,5,1,10};
  init_segment_tree(7); build_segment_tree(RANGE_MIN, A, 1, 0, 6);
  printf("%d\n", query(RANGE_MIN, A, 1, 0, 6, 1, 3)); // answer is index 2
  return 0;
}
```

Exercise 1: Draw a segment tree of this array A = {10, 2, 47, 3, 7, 9, 1, 98, 21, 37} and answer RMQ(1, 7) and RMQ(3, 8)!

Exercise 2: Using the same 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? (See Section 3.4).

Exercise 3: The Segment Tree code shown above lacks update operation. Add the $O(\log n)$ `update` function to update the value of a certain segment in the Segment Tree!

Programming exercises that use data structures with our own libraries:

- Graph (simple one, many more in Section 4)
  1. UVa 291 - The House of Santa Claus (simple backtracking)
  2. UVa 10928 - My Dear Neighbours (count 'out degrees')

- Union-Find Disjoint Sets
  1. UVa 459 - Graph Connectivity (also solvable with 'flood fill' in Section 4.2)
  2. UVa 793 - Network Connections (trivial)
  3. UVa 912 - Live From Mars
  4. UVa 10158 - War
  5. UVa 10301 - Rings and Glue (with Computational Geometry, see Chapter 7)
  6. UVa 10369 - Arctic Networks (also solveable with MST, see Section 4.4)
  7. UVa 10505 - Montesco vs Capuleto
  8. UVa 10583 - Ubiquitous Religions (count remaining disjoint sets after unions)
  9. UVa 10608 - Friends (search set with largest element)
  10. UVa 11503 - Virtual Friends (maintain set attribute (size) in representative item)
- Segment Tree
  1. UVa 11235 - Frequent Values (Range Maximum Query)
  2. UVa 11297 - Census (2-D Segment Tree/Quad Tree)
  3. UVa 11402 - Ahoy, Pirates! (Requires updates to the Segment Tree)
  4. LA 2191 - Potentiometers (Dhaka06)
  5. LA 3294 - The Ultimate Bamboo Eater (Dhaka05) (2-D Segment Tree++)
  6. LA 4108 - Skyline (Singapore07)

2.4 Chapter Notes

Basic data structures mentioned in Section 2.2 can be found in almost every data structure and algorithm textbooks and references to their libraries are available online in:
`http://www.cppreference.com` and `http://java.sun.com/javase/6/docs/api`.

Extra references for data structures mentioned in Section 2.3 are as follow: For Graph data structure, a good textbook is [21]. For Union-Find Disjoint Sets, see Chapter 21 of [4]. For Segment Tree and other geometric-related data structures, see [6].

With more experience, and especially by looking at TopCoder's codes, you will master more tricks in using these data structures. This point is hard to teach. Thus, please spend some time to explore the sample codes listed in this book.

The discussion about string-specific data structures (**Suffix Tree** and **Suffix Array**) is deferred until Section 6.4. Yet, there are still many other data structures that we cannot cover in this book. If you want to win a programming contest, mastering more than what we present in this book will increase your chance to do better in contests, e.g. **AVL Tree**, **Red Black Tree**, **Splay Tree** which are useful for certain contest problems where you need to implement and augment a balanced BST; **Fenwick (Binary Indexed) Tree** which can be used to implement cumulative frequency tables; **Quad Tree** for partitioning 2-D space; etc.

Notice that many of the efficient data structures shown in this book have the spirit of Divide and Conquer (discussed in Section 3.2).

There are approximately **43 programming exercises** discussed in this chapter.

Chapter 3

Problem Solving Paradigms

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

This chapter highlights four problem solving paradigms commonly used to attack problems in programming contests, namely Complete Search, Divide & Conquer, Greedy, and Dynamic Programming. Mastery of all these problem solving paradigms will help contestants to attack each problem with the appropriate 'tool', rather than 'hammering' every problem with brute-force solution... which is clearly not competitive. Our advice before you start reading: Do not just remember the solutions for the problems presented in this chapter, but remember the way, the spirit of solving those problems!

3.1 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 in bid 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 $\{1, 2, 3, .., N\}$, 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 $O(N)$ loop).

In ICPC, Complete Search should be the first considered solution as it is usually easy to come up with the solution and code/debug it. 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.

Sometimes, running Complete Search on small instances of a hard problem can give us some patterns from the output that can be exploited to design faster algorithm. The Complete Search solution can also act as a verifier for faster but non-trivial algorithms (but only on small instances).

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

3.1.1 Examples

We show two examples of Complete Search: one that is implemented iteratively and one that is implemented recursively (backtracking). We also mention a few optimization tricks to make some 'impossible' cases become possible.

1. 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 numerals is allowed to be zero, e.g. `79546 / 01283 = 62; 94736 / 01528 = 62`.

   A quick analysis shows that `fghij` can only be 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 1: What is the advantage of iterating through all possible `fghij` and not `abcde`?

   Exercise 2: Does a 10! algorithm that permutes `abcdefghij` work?

2. Recursive Backtracking: UVa 750 - 8 Queens Chess Problem

   Abridged problem statement: In chess (with a standard 8x8 board), it is possible to place eight queens on the board so that no queen can be taken by any other. Write a program that will determine *all* such possible arrangements given the initial position of one of the queens (i.e. coordinate (a, b) in the board must contain a queen).

   A naïve solution tries all $8^8 \approx 17M$ possible arrangements of 8 queens in an 8x8 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 more.

   We know that no two queens can share the same column, thus we can simplify the problem to finding valid permutation of 8! row positions `row[i]` stores the row position of 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 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 diagonals. 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)`. Go ahead and verify this formula by placing two queens randomly but on the same diagonal.

   A recursive backtracking solution will then place 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 constrain, i.e. `row[b] == a`. This solution is Accepted.

   We provide our code in the next page below. If you have never code a recursive backtracking solution before, please scrutinize it and perhaps re-code it using your own coding style. For example, you may want to use C++ STL <`algorithm`>: `next_permutation` instead.

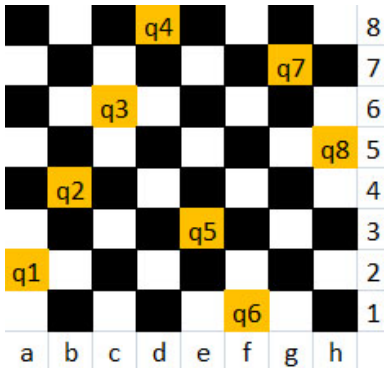


Figure 3.1: One Solution for 8-Queens Problem: {2, 4, 6, 8, 3, 1, 7, 5}

```
/* 8 Queens Chess Problem */
#include <iostream>
#include <string.h>
#include <math.h>
using namespace std;

int x[9], TC, a, b, lineCounter; // it is ok to use global variables in competitive programming

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

void NQueens(int queen) {
  for (int row = 1; row <= 8; row++)
    if (place(queen, row)) { // if can place this queen at this row?
      x[queen] = row; // put this queen in this row
      if (queen == 8 && x[b] == a) { // a candidate solution & (a, b) has 1 queen
        printf("%2d      %d", ++lineCounter, x[1]);
        for (int j = 2; j <= 8; j++) printf(" %d", x[j]);
        printf("\n");
      }
      else
        NQueens(queen + 1); // recursively try next position
}   }

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

3.1.2 Tips

The biggest gamble in writing a Complete Search solution is whether it will be able to pass the Time Limit. If it 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 rather than painfully redo 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 for passing the Time Limit.

Tip 1: Generating versus Filtering

Programs that generate lots of candidate solutions and then choose the ones that are correct (or remove the incorrect ones) are called 'filters' – recall the naïve 8-queens solver with 8^8 time complexity. Those that hone in exactly to the correct answer without any false starts are called 'generators' – recall the improved 8-queens solver with 8! complexity plus diagonal checks.

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 (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. 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 branches will never lead to a valid solution. Thus we can prune these branches 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) solutions as there are rotations and reflections symmetries in this problem [35]. 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.

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.

Again using the 8-queens problem above. If we know that there are only 92 solutions, then we can create a 2-dimensional array `int solution[92][8]` and then fill it with all 92 valid permutations of 8 queens row positions! That's it, we create a generator program (which takes some runtime) to fill this 2-D array `solution`, but afterwards, we generate a new program and submit the code that just prints out the correct permutations with 1 queen at (a, b) (very fast).

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

Tip 5: Try Solving the Problem Backwards

Surprisingly, some contest problems seem far easier when they are solved backwards than when they are solved using a frontal attack. Be on the lookout for processing data in reverse order or building an attack that looks at the data in some order other than *the obvious*.

This tip is best shown using an example: UVa 10360 - Rat Attack. Abridged problem description: Imagine a 2-D array (up to 1024 x 1024) containing rats. There are $n \leq 20000$ rats at some cells, determine which cell (x, y) should be gas-bombed so that the number of rats killed in square box (x - d, y - d) to (x + d, y + d) is maximized. The value d is the power of the gas-bomb (d is up to 50), see Figure 3.2.

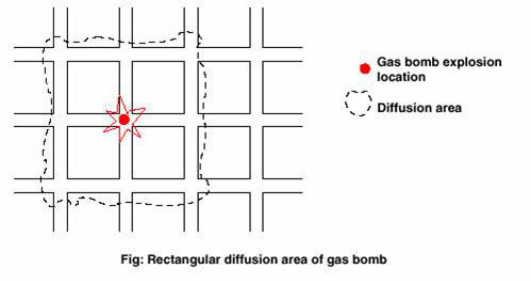


Figure 3.2: UVa 10360 - Rat Attack Illustration with d = 1

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. Clearly TLE!

Second option is to attack this problem backwards: Create an array `int killed[1024][1024]`. For each n rat population at coordinate (x, y), add the value of array `killed[i][j]` with the number of rats in (x, y) that will be killed if a bomb is placed in (i, j) and (i, j) is within the square-bombing radius (i.e. $|i - x| \leq d$ and $|j - y| \leq d$). 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 $O(n^2)$ operations. This backwards approach only requires $20000 \times 50^2 + 20000^2 = 51M$ operations for the worst test case ($n = 20000, d = 50$) and approximately 51 times faster than the frontal attack!

Tip 6: Optimizing 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.
2. Use the expected $O(n \log n)$ but cache-friendly quicksort (built-in in C++ STL `sort` as part of 'introsort') rather than the true $O(n \log n)$ but not (cache) memory friendly mergesort.
3. Access a 2-D array in a row major fashion (row by row) rather than column by column.
4. Bitwise manipulation on integer is faster than using an array of bits (see Section 3.4.3), etc.
5. Use STL <`bitset`> rather than `vector<bool>` for Sieve of Eratosthenes (see Section 5.3.1).

6. Declare a bulky data structure just once by setting it to have global scope, so you do not have to pass the structure as function arguments.
7. Allocate memory just once, according to the largest possible input in the problem description, rather than re-allocating it for every test case in a multiple-input problem.

Browse the Internet or reference books to find more information on how to speed up your code.

Tip 7: Use Better Data Structure & Algorithm :)

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

Programming Exercises solvable using Complete Search:

- Iterative
  1. UVa 154 - Recyling (try all combinations)
  2. UVa 441 - Lotto (6 nested loops!)
  3. UVa 639 - Don't Get Rooked (generate 2^{16} possible combinations, prune invalid ones)
  4. UVa 725 - Division (elaborated in this section)
  5. UVa 10360 - Rat Attack (this problem is also solvable using 1024^2 DP range sum)
  6. UVa 10662 - The Wedding (3 nested loops!)
  7. UVa 11242 - Tour de France (iterative complete search + sorting)
  8. UVa 11804 - Argentina (5 nested loops!)
- Recursive Backtracking
  1. UVa 193 - Graph Coloring (Maximum Independent Set)
  2. UVa 222 - Budget Travel (input not large)
  3. UVa 524 - Prime Ring Problem (also see Section 5.3.1)
  4. UVa 624 - CD (input size is small, use backtracking; also solve-able with DP)
  5. UVa 628 - Passwords (backtracking)
  6. UVa 729 - The Hamming Distance Problem (backtracking)
  7. UVa 750 - 8 Queens Chess Problem (solution already shown in this section)
  8. UVa 10285 - Longest Run on a Snowboard (backtracking, also solve-able with DP)
  9. UVa 10496 - Collecting Beepers (small TSP instance)
  10. LA 4793 - Robots on Ice (World Finals Harbin10, recommended problem for practice)

Problem I - 'Robots on Ice' in the recent ACM ICPC World Final 2010 can be viewed as a 'tough test on pruning strategy'. The problem is simple: Given an M x N board with 3 check-in points {A, B, C}, find a Hamiltonian path of length (M x 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 tour, respectively. Constraints: $2 \leq M, N \leq 8$.

A naïve recursive backtracking algorithm will get TLE. To speed up, we must prune the search space if: 1). it does not hit the appropriate target check point at 1/4, 1/2, or 3/4 distance; 2). it hits target check point earlier than the target time; 3). it will not be able to reach the next check point on time from the current position; 4). it will not be able to reach final point (0, 1) as the current path blocks the way. These 4 pruning strategies are sufficient to solve LA 4793.

3.2 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 chapters in this book: various sorting algorithms like 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, and Segment Tree in Section 2.2.2 & 2.3.3, also has the spirit of Divide & Conquer.

3.2.1 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. [14]) 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 is 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 query, 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 and apply binary search. The pre-requisite to run binary search algorithm – a *static sorted array (or vector)* – can also be found in other uncommon data structure, as in the root-to-leaf path on a structured tree below.

Binary Search on Uncommon Data Structure (Thailand ICPC National Contest 2009)

Problem in short: given a weighted (family) tree of N vertices up to $N \leq 80K$ with a special trait: *vertex values are increasing from root to leaves*. Find the ancestor vertex closest to root from a starting vertex v that has weight at least P. There are up to $Q \leq 20K$ such queries.

Naïve solution is to do this linear $O(N)$ scan per query: Start from a given vertex v, then move up the family tree until we hit the first ancestor with value $< P$. In overall, as there are Q queries, this approach runs in $O(QN)$ and will get TLE as $N \leq 80K$ and $Q \leq 20K$.

A better solution is to store all the $20K$ queries first. Then traverse the family tree *just once* from root using $O(N)$ Depth First Search (DFS) algorithm (Section 4.2). Search for some non-existent value so that DFS explores the *entire* tree, building a partial root-to-leaf *sorted* array as it goes – this is because the vertices in the root-to-leaf path have increasing weights. Then, for each

vertex asked in query, perform a $O(\log N)$ **binary search**, i.e. `lower_bound`, along the current path from root to that vertex to get ancestor closest to root with weight at least P. 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 usage of binary search in finding items in a static sorted array. However, the binary search **principle** can also be used to find the root of a function that may be difficult to compute mathematically.

Sample problem: You want to buy a car using loan and want to pay 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 every unpaid money 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 $d = 576$, $m = 2$, $v = 1000$, and $i = 10\%$. After one month, your loan becomes $1000 \times (1.1) - 576 = 524$. After two months, your loan becomes $524 \times (1.1) - 576 \approx 0$.

But if we are only given $m = 2$, $v = 1000$, and $i = 10\%$, how to determine that $d = 576$? In another words, find the root d such that loan payment function $f(d, 2, 1000, 10) \approx 0$. The *easy* way is to run the bisection method[2]. We pick a reasonable range as the starting point. In this case, we want to find d within range $[a \dots b]$. $a = 1$ 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 $(1 + i\%) \times v = (1 \times 10) \times 1000 = 1100$ dollars after one month. Then, we apply bisection method to obtain d as follows:

- If we pay $d = (a + b)/2 = (1 + 1100)/2 = 550.5$ dollars per month, then we undershoot by 53.95 dollars after two months, so we know that we must *increase* the monthly payment.
- If we pay $d = (550.5 + 1100)/2 = 825.25$ dollars per month, then we overshoot by 523.025 dollars after two months, so we know that we must *decrease* the payment.
- If we pay $d = (550.5 + 825.25)/2 = 687.875$ dollars per month, then we overshoot by 234.5375 dollars after two months, so we know that we must *decrease* the payment.
- ... **few** logarithmic iterations after, to be precise, after $O(\log_2((b - a)/\epsilon))$ iterations where ϵ is the amount of error that we can tolerate.
- Finally, if we pay $d = 576.190476...$ dollars per month, then we manage to finish the payment (the error is now less than ϵ) after two months, so we know that $d = 576$ is the answer.

For bisection method to work[3], we must ensure that the function values of the two extreme points in the initial Real range $[a \dots b]$, i.e. $f(a)$ and $f(b)$ have opposite signs (true in the problem above). Bisection method in this example only takes $\log_2 1099/\epsilon$ tries. Using a small ϵ = 1e-9, this is just $\approx$ 40 tries. Even if we use an even smaller ϵ = 1e-15, we still just need $\approx$ 60 tries[4]. Bisection method is more efficient compared to linearly trying each possible value of $d = [1..1100]/\epsilon$.

[2]We use the term 'binary search principle' as a divide and conquer technique that involve halving the range of possible answer. We use the term 'binary search algorithm' (finding index of certain item in sorted array) and 'bisection method' (finding root of a function) as instances of this principle.

[3]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.

[4]Thus some competitive programmers choose to do 'loop 100 times' which guarantees termination instead of testing whether the error is now less than ϵ as some floating point errors may lead to endless loop.

Binary Search the Answer

Binary Search 'the Answer' is another problem solving strategy that can be quite powerful. This strategy is shown using UVa 714 - Copying Books below.

In this problem, you are given m 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 divide 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. The time needed to make a copy of all the books is determined by the scriber who was assigned the most work. The task is to minimize the maximum number of pages assigned to a single scriber.

There exist Dynamic Programming solution for this problem, but this problem can already be solved by guessing the answer in binary search fashion! Suppose $m = 9$, $k = 3$ and $p_1, p_2, \ldots, p_9$ are 100, 200, 300, 400, 500, 600, 700, 800, and 900, respectively.

If we guess $ans = 1000$, 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 as: {100, 200, 300, 400} for scribe 1, {500} for scribe 2, {600} for scribe 3, but we have 3 books {700, 800, 900} unassigned. The answer must be at least 1000.

If we guess $ans = 2000$, then we greedily assign the jobs as: {100, 200, 300, 400, 500} for scribe 1, {600, 700} for scribe 2, and {800, 900} for scribe 3. We still have some slacks, i.e. scribe 1, 2, and 3 still have {500, 700, 300} unused potential. The answer must be at most 2000.

This ans is binary-searchable between $lo = 1$ (1 page) and $hi = p_1 + p_2 + \ldots + p_m$ (all pages).

Programming Exercises for problems solvable using Divide and Conquer:

1. UVa 679 - Dropping Balls (like Binary Search)
2. UVa 714 - Copying Books (Binary Search + Greedy)
3. UVa 957 - Popes (Complete Search + Binary Search: `upper_bound`)
4. UVa 10077 - The Stern-Brocot Number System (Binary Search)
5. UVa 10341 - Solve It (Bisection Method)
6. UVa 10369 - Arctic Networks (solvable using Kruskal's for MST too – Section 4.4)
7. UVa 10474 - Where is the Marble?
8. UVa 10611 - Playboy Chimp (Binary Search)
9. UVa 11262 - Weird Fence (Binary Search + Bipartite Matching, see Section 4.9.3)
10. LA 2565 - Calling Extraterrestrial Intelligence Again (Kanazawa02) (BSearch + Math)
11. LA 2949 - Elevator Stopping Plan (Guangzhou03) (Binary Search + Greedy)
12. LA 3795 - Against Mammoths (Tehran06)
13. LA 4445 - A Careful Approach (Complete Search + Bisection Method + Greedy)
14. IOI 2006 - Joining Paths
15. IOI 2009 - Mecho (Binary Search + BFS)
16. My Ancestor - Thailand ICPC National Contest 2009 (Problem set by: Felix Halim)
17. See Section 7.5 for 'Divide & Conquer for Geometry Problems'

3.3 Greedy

An algorithm is said to be greedy if it makes locally optimal choice at each step with the hope of 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 [4], a problem must exhibit two things in order for a greedy algorithm to work for it:

1. It has optimal sub-structures.

   Optimal solution to the problem contains optimal solutions to the sub-problems.

2. It has a greedy property (remark: 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.

3.3.1 Classical Example

Suppose we have a large number of coins with different denominations, i.e. 25, 10, 5, and 1 cents. We want to make change with the *least number of coins used*. 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),
   b. To make 7 cents, we have to use 5+1+1 (3 coins), 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 other strategy than this will not lead to optimal solution.

However, this greedy algorithm does *not* work for *all* sets of coin denominations, e.g. {1, 3, 4} 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 3.4.2.

There are many other classical examples of greedy algorithms in algorithm textbooks, for example: Kruskal's for Minimum Spanning Tree (MST) problem – Section 4.4, Dijkstra's for Single-Source Shortest Paths (SSSP) problem – Section 4.5, Greedy Activity Selection Problem [4], Huffman Codes [4], etc.

3.3.2 Non Classical Example

Today's contest problems usually do not ask for solution of trivial and classical greedy problems. Instead, we are presented with novel ones that requires creativity, like the one shown below.

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 3.3 for visual explanation.

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.

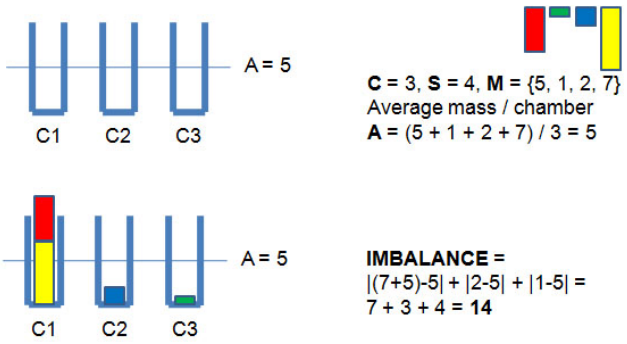


Figure 3.3: Visualization of UVa 410 - Station Balance

This problem can be solved using a greedy algorithm. But first, we have to make several observations. If there exists an empty chamber, at least one chamber with 2 specimens must be moved to this empty chamber! Otherwise the empty chambers contribute too much to IMBALANCE! See Figure 3.4.

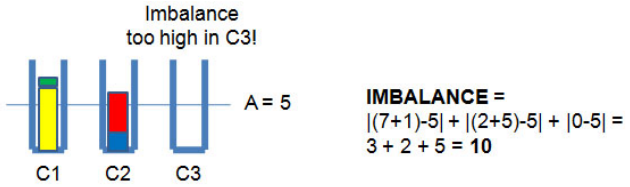


Figure 3.4: UVa 410 - Observation 1

Next observation: If $S > C$, then $S - C$ specimens must be paired with one other specimen already in some chambers. The Pigeonhole principle! See Figure 3.5.

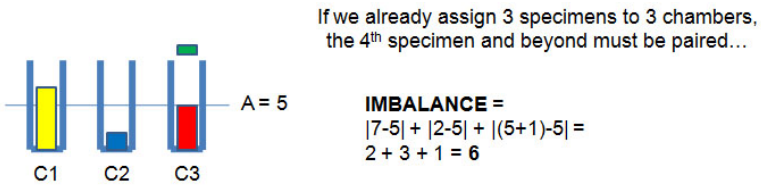


Figure 3.5: UVa 410 - Observation 2

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, $M = \{5, 1, 2, 7, 0, 0\} \rightarrow \{0, 0, 1, 2, 5, 7\}$.

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 3.6.

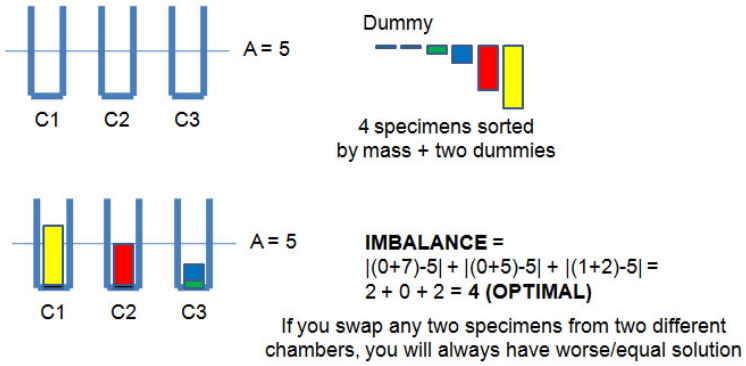


Figure 3.6: UVa 410 - Greedy Solution

To come up with this way of thinking is hard to teach but can be gained from experience! One tip from this example: If no obvious greedy strategy seen, try to sort the data first or introduce some tweaks and see if a greedy strategy emerges.

3.3.3 Remarks About Greedy Algorithm in Programming Contests

Using Greedy solutions in programming contests is usually risky. A greedy solution normally will not encounter TLE response, as it is lightweight, but tends to get WA response. Proving that a certain problem has optimal sub-structure and greedy property in contest time may be time consuming, so a competitive programmer usually do this:

He will look at the input size. If it is 'small enough' for the time complexity of either Complete Search or Dynamic Programming (see Section 3.4), he will use one of these approaches as both will ensure correct answer. He will *only* use Greedy solution if he knows for sure that the input size given in the problem is too large for his best Complete Search or DP solution.

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

Programming Exercises solvable using Greedy (hints omitted):

1. UVa 410 - Station Balance (elaborated in this section)
2. UVa 10020 - Minimal Coverage
3. UVa 10340 - All in All
4. UVa 10440 - Ferry Loading II
5. UVa 10670 - Work Reduction
6. UVa 10763 - Foreign Exchange
7. UVa 11054 - Wine Trading in Gergovia
8. UVa 11292 - Dragon of Loowater
9. UVa 11369 - Shopaholic

In this section, we want to highlight another problem solving trick called: *Decomposition*!

While there are only 'few' basic algorithms used in contest problems (most of them are covered in this book), harder problems may require a *combination* of two (or more) algorithms for their solution. For such problems, try to decompose parts of the problems so that you can solve the different parts independently. We illustrate this decomposition technique using a recent top-level programming problems that combines *three* problem solving paradigms that we have just learned: Complete Search, Divide & Conquer, and Greedy!

ACM ICPC World Final 2009 - Problem A - A Careful Approach

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, b_i, which give 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 $0 \leq a_i \leq b_i \leq 1440$. In this problem, 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 'greedy activity selection' problem [4]?
3. Print the answer split into minutes and seconds, rounded to the closest second.
See Figure 3.7 for illustration: line = the time window of a plane; star = its landing schedule.

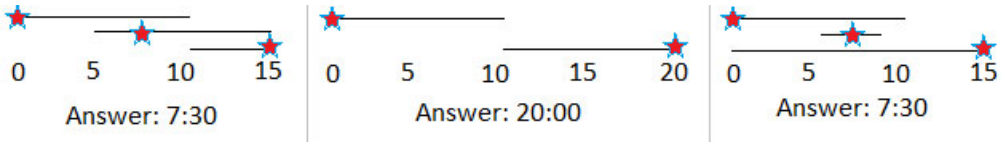


Figure 3.7: Illustration for ACM ICPC WF2009 - A - A Careful Approach

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** portion of the problem which can be easily solved using C++ STL `next_permutation`.

Now, for each specific landing order, we want to know the largest possible landing window. Suppose we use 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 a **Greedy Algorithm**.

A window length L that is too long/short will overshoot/undershoot `b[last plane]`, so we have to decrease/increase L. We can binary search the answer L – **Divide & Conquer**. As we only want the answer rounded to nearest integer, stopping binary search when error $\epsilon \leq$ 1e-3 is enough. For more details, please study our AC source code shown on the next page.

```
/* World Final 2009, A - A Careful Approach, LA 4445 (Accepted) */
#include <algorithm>
#include <cmath>
#include <stdio.h>
using namespace std;

int i, n, caseNo = 1, order[8];
double a[8], b[8], timeGap, maxTimeGap; // timeGap is the variable L mentioned in text

double greedyLanding() { // with certain landing order, and certain timeGap,
                         // 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 + timeGap;
    if (targetLandingTime <= b[order[i]])
      // this aircraft can land, greedily choose max of a[order[i]] or targetLandingTime
      lastLanding = max(a[order[i]], targetLandingTime);
    else
      return 1; // returning positive value will force binary search to reduce timeGap
  }
  // returning negative value will force binary search to increase timeGap
  return lastLanding - b[order[n - 1]];
}

int main() {
  while (scanf("%d", &n), n) { // 2 <= n <= 8
    for (i = 0; i < n; i++) {
      scanf("%lf %lf", &a[i], &b[i]); // [ai, bi] is the interval where plane i can land safely
      a[i] *= 60; b[i] *= 60; // originally in minutes, convert to seconds
      order[i] = i;
    }

    maxTimeGap = -1; // variable to be searched for
    do { // permute plane landing order, 8!
      double lowVal = 0, highVal = 86400; // min 0s, max 1 day = 86400s
      timeGap = -1; // start with 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(lowVal - highVal) >= 1e-3) { // binary search timeGap, ERROR = 1e-3 is OK here
        timeGap = (lowVal + highVal) / 2.0;    // we just want the answer to be rounded to nearest int
        double retVal = greedyLanding(); // round down first
        if (retVal <= 1e-2) // must increase timeGap
          lowVal = timeGap;
        else // if (retVal > 0) // infeasible, must decrease timeGap
          highVal = timeGap;
      }
      maxTimeGap = max(maxTimeGap, timeGap); // get the max over all permutations
    }
    while (next_permutation(order, order + n)); // keep trying all permutations
    // another way for rounding is to use printf format string: %.0lf:%0.2lf
    maxTimeGap = (int)(maxTimeGap + 0.5); // round to nearest second
    printf("Case %d: %d:%0.2d\n", caseNo++, (int)(maxTimeGap / 60), (int)maxTimeGap % 60);
} } // return 0;
// Challenge: rewrite this code to avoid double!
```

3.4 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 before continuing. Plus, get yourself ready to see lots of recursions and recurrence relations!

3.4.1 DP Illustration Using UVa 11450 - Wedding Shopping

Abridged problem statement: Given different models for each garment (e.g. 3 shirts, 2 belts, 4 shoes, ...), *buy one model of each garment*. As the budget is *limited*, we cannot spend more money than the budget, but we want to spend *the maximum possible*. But it is also possible that we cannot buy one model of each garment due to that small amount of budget.

The input consist 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, there are information of the C garments. For a garment_id $\in [0 \ldots C\text{-}1]$, we know an integer $1 \leq K \leq 20$ which indicates the number of different models for that garment_id, followed by K integers indicating the price of each model $\in [1 \ldots K]$ of that garment_id.

The output should consist of one integer that indicates the maximum amount of money necessary to buy one element of each garment *without exceeding the initial amount of money*. If there is no solution, print "no solution".

For example, if the input is like this (test case A):
M = 20, C = 3
3 models of garment_id 0 → 6 4 <u>8</u> // see that the prices are not sorted in input
2 models of garment_id 1 → 5 <u>10</u>
4 models of garment_id 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 we also have (6+10+3) and (4+10+5).

However, if the input is like this (test case B):
M = **9 (very limited budget)**, C = 3
3 models of garment_id 0 → 6 4 8
2 models of garment_id 1 → 5 10
4 models of garment_id 2 → 1 5 3 5

Then the answer is "no solution" as buying all the cheapest models (4+5+1) = 10 is still $> M$.

Approach 1: Complete Search (Time Limit Exceeded)

First, let's see if Complete Search (backtracking) can solve this problem: Start with `money_left = M` and `garment_id = 0`. Try all possible models in that `garment_id = 0` (max 20 models). If model i is chosen, then subtract `money_left` with model i's price, and then recursively do the same process to `garment_id = 1` (also can go up to 20 models), etc. Stop if the model for the last `garment_id = C - 1` has been chosen. If `money_left < 0` before we reach the last `garment_id`, prune this partial solution. Among all valid combinations, pick one that makes `money_left` as close to 0 as possible yet still ≥ 0. This maximizes the money spent, which is (`M - money_left`).

We can formally define these Complete Search recurrences:

1. if `money_left < 0` (i.e. money goes negative),
`shop(money_left, garment_id) = INVALID` (in practice, we can return a large negative value)
2. if a model from the last `garment_id` has been bought (i.e. a candidate solution),
`shop(money_left, garment_id) = M - money_left` (this is the actual money spent)
3. in general case, for all model in $[1 \ldots K]$ of current `garment_id`

```
shop(money_left, garment_id) =
    max(shop(money_left - price[garment_id][model], garment_id + 1))
```

We want to maximize this value (Recall that 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_id 0 have up to 20 choices; garment_id 1 *also* have up to 20 choices; ...; and the last garment_id 19 ***also*** have up to 20 choices. Therefore, Complete Search like this runs in $20 \times 20 \times ... \times 20$ of total 20 times in the worst case, i.e. 20^{20} = a **very very large** number. If we *only* know Complete Search, there is no way we can solve this problem.

Approach 2: Greedy (Wrong Answer)

Since we want to maximize the budget spent, why don't we take the most expensive model in each `garment_id` which still fits our budget? For example in test case A above, we choose the most expensive model 3 of `garment_id = 0` with cost 8 (`money_left = 20-8 = 12`), then choose the most expensive model 2 of `garment_id = 1` with cost 10 (`money_left = 12-10 = 2`), and then for `garment_id = 2`, we can only choose model 1 with cost 1 as `money_left` does not allow us to buy other models with cost 3 or 5. This greedy strategy 'works' for test cases A+B above and produce the same optimal solution (8+10+1) = 19 and "no solution", respectively. It also runs very fast, which is $20 + 20 + ... + 20$ of total 20 times = 400 operations in the worst case.

But greedy does not work for many other cases. This test case below is a counter-example:
M = 12, C = 3
3 models of garment_id 0 → 6 4 **8**
2 models of garment_id 1 → 5 10
4 models of garment_id 2 → 1 5 3 5

Greedy strategy selects model 3 of `garment_id = 0` with cost **8** (`money_left = 12-8 = 4`), thus we do not have enough money to buy any model in `garment_id = 1` and wrongly reports "no solution". The optimal solution is actually (4+5+3 = 12), which use all our budget.

Approach 3: Top-Down DP (Accepted)

To solve this problem, we have to use DP. Let's see the key ingredients to make DP works:

1. This problem has optimal sub-structures.
   This is shown in Complete Search recurrence 3 above: solution for the sub-problem is part of the solution of the original problem. Although optimal sub-structure are the same ingredient to make a Greedy Algorithm work, this problem lacks the 'greedy property' ingredient.
2. This problem has overlapping sub-problems.
   This is the key point of DP! The search space is actually not as big as 20^{20} analyzed in Complete Search discussion above as **many** sub-problems are actually overlapping!

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

So, how many *distinct* sub-problems (a.k.a. **states**) are there in this problem? The answer is, only $201 \times 20 = 4{,}020$. As there only there are only 201 possible `money_left` (from 0 to 200, inclusive) and 20 possible `garment_id` (from 0 to 19, inclusive). Each of the sub-problem just need to be computed *only once*. If we can ensure this, we can solve this problem much faster.

Implementation of this DP solution is surprisingly simple. If we already have the recursive backtracking (the recurrence relation shown previously), we can implement **top-down** DP by doing these few additional steps:

1. Initialize a DP 'memo' table with dummy values, e.g. '-1'.
   The dimension of the DP table must be the size of distinct sub-problems.
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, $O(1)$.
   (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 is fast.

Analyzing DP solution is easy. If it has M states, then it requires at least $O(M)$ memory space. If filling a cell in this state requires $O(k)$ steps, then the overall time complexity is $O(kM)$. 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_id`). Thus the time complexity is $4020 \times 20 = 80400$, which is very manageable. We show our code below as an illustration, especially for those who have never coded a top-down DP algorithm before.

```
/* UVa 11450 - Wedding Shopping */
#include <iostream>
#include <algorithm>
#include <string.h>
using namespace std;

int M, C, K, price[25][25]; // price[garment_id (<= 20)][model (<= 20)]
int memo[210][25]; // dp table memo[money_left (<= 200)][garment_id (<= 20)]

int shop(int money_left, int garment_id) {
  if (money_left < 0)
    return -1000000000; // fail, return a large negative number (1B)
  if (garment_id == C) // we have bought last garment
    return M - money_left; // done, return this value
  if (memo[money_left][garment_id] != -1) // if this state has been visited before
    return memo[money_left][garment_id]; // simply return it
  int max_value = -1000000000;
  for (int model = 1; model <= price[garment_id][0]; model++) // try all possible models
    max_value = max(max_value, shop(money_left - price[garment_id][model], garment_id + 1));
  return memo[money_left][garment_id] = max_value;  // assign max_value 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", &K);
      price[i][0] = K; // to simplify coding, we store K in price[i][0]
      for (j = 1; j <= K; j++)
        scanf("%d", &price[i][j]);
    }

    memset(memo, -1, sizeof memo); // initialize DP memo table
    score = shop(M, 0); // start the top-down DP
    if (score < 0)
      printf("no solution\n");
    else
      printf("%d\n", score);
  }
} // return 0;
```

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 a.k.a. the 'tabular method'. The steps to build bottom-up DP are like this:

1. Identify the Complete Search recurrence as with top-down DP above.
2. Initialize some parts of the DP table with known initial values.
3. Determine how to fill the rest of the DP table based on the Complete Search recurrence, usually involving one or more nested loops to do so.

For UVa 11450 above, we can write the bottom-up DP as follow. For clarity of this discussion, please see Figure 3.8 which illustrates test case A above.

First, set up a boolean matrix `can_reach[money_left][garment_id]` of size 201 × 20. At first, only the cells reachable by buying any of the models of `garment_id = 0` are true. See Figure 3.8, leftmost, where only rows '20-6 = 14', '20-4 = 16', and '20-8 = 12' in column 0, are true.

Then, we loop through from second garment until the last garment. We set `can_reach[a][b]` to be true if it is possible to reach this state from any states in previous column, i.e. from state `can_reach[a + price of any model of previous garment_id][b - 1]`. See Figure 3.8, middle, where for example, `can_reach[11][1]` can be reached from `can_reach[11 + 5][0]` by buying a model with cost 5 in `garment_id = 1`; `can_reach[2][1]` can be reached from `can_reach[2 + 10][0]` by buying a model with cost 10 in `garment_id = 1`; etc.

Finally, the answer can be found in the last column. Find the cell in that column nearest to index 0 that is set to be true. In Figure 3.8, rightmost, the cell `can_reach[1][2]` is the answer. This means that we can somehow reach this state (`money_left = 1`) by buying combination of various garment models. The final answer is actually `M - money_left`, 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.

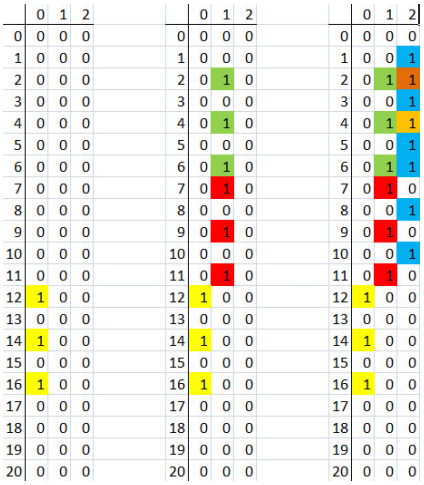


Figure 3.8: UVa 11450 - Bottom-Up DP Solution

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

```
#include <iostream>
#include <string.h>
using namespace std;

int main() {
  int i, j, l, TC, M, C, K, price[25][25]; // price[garment_id (<= 20)][model (<= 20)]
  bool can_reach[210][25]; // can_reach table[money_left (<= 200)][garment_id (<= 20)]
                           // question: is 2nd dimension (model) needed? M = (201*20) -> (201) only?
  scanf("%d", &TC);        // can we compute the solution by just maintaining 2 most recent columns?
  while (TC--) {           // hint: DP-on-the-fly (a.k.a space saving trick)
    scanf("%d %d", &M, &C);
    for (i = 0; i < C; i++) {
      scanf("%d", &K);
      price[i][0] = K; // to simplify coding, we store K in price[i][0]
      for (j = 1; j <= K; 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_id

    // Challenge: this loop is written in column major, rewrite it in row major!
    // See Tips 6.3 in Section 3.1.2
    for (j = 1; j < C; j++) // for each remaining garment
      for (i = 0; i < M; i++) if (can_reach[i][j - 1]) // if can reach this state
        for (l = 1; l <= price[j][0]; l++) if (i - price[j][l] >= 0) // flag the rest
          can_reach[i - price[j][l]][j] = true; // as long as it is feasible

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

    if (i == M + 1) // nothing in this last column has its bit turned on
      printf("no solution\n");
    else
      printf("%d\n", M - i);
} } // return 0;
```

Top-Down versus Bottom-Up DP

As you can see, the way the bottom-up DP table is filled is not as intuitive as the top-down DP as it requires some 'reversals' of the signs in Complete Search recurrence that we have developed in previous sections. However, we are aware that some programmers actually feel that the bottom-up version is more intuitive. The decision on using which DP style is in your hand. To help you decide which style that you should take when presented with a DP solution, we present the trade-off comparison between top-down and bottom-up DP in Table 3.1.

Top-Down	**Bottom-Up**
Pro: 1. It is a natural transformation from normal Complete Search recursion 2. Compute sub-problems only when necessary (sometimes this is faster)	Pro: 1. Faster if many sub-problems are revisited as there is no overhead from recursive calls 2. Can save memory space with DP 'on-the-fly' technique (see comment in code above)
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 hard problems	Cons: 1. For some programmers who are inclined with recursion, this may be not intuitive 2. If there are M states, bottom-up DP visits and fills the value of *all* these M states

Table 3.1: DP Decision Table

3.4.2 Several Classical DP Examples

There are many other *well-known* problems with efficient DP solutions. We consider these problems as classic and therefore must be known by everyone who wish to do well in ICPC or IOI!

Longest Increasing Subsequence (LIS)

Problem: Given a sequence {X[0], X[1], ..., X[N-1]}, determine its Longest Increasing Subsequence (LIS)[5] – as the name implies. Take note that 'subsequence' is not necessarily contiguous.

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

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) = ans, computed with a loop below:

```
int ans = 1;
for (int j = 0; j < i; j++) // O(n)
  if (X[i] > X[j]) // if we can extend LIS(j) with i
    ans = max(ans, 1 + LIS(j))
```

[5]There are other variants of this problem: Longest *Decreasing* Subsequence, Longest *Non Increasing/Decreasing* Subsequence, and the $O(n \log k)$ solution by utilizing the fact that the LIS is sorted and binary-searchable. See `http://en.wikipedia.org/wiki/Longest_increasing_subsequence` for more details. Note that increasing subsequences can be modeled as a Directed Acyclic Graph (DAG). Thus finding LIS is equivalent to finding longest path in DAG.

The answer is the highest value of LIS(k) for all k in range [0 ... N-1].

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 3.9: Longest Increasing Subsequence

In Figure 3.9, we see that:
LIS(0) is 1, the number -7 itself.
LIS(1) is now 2, as we can form {-7, 10} of length 2.
LIS(2) is also 2, as we can form {-7, 9}, but not {-7, 10} + {9} as it is non increasing.
LIS(3) is also 2, we can only form {-7, 2}. {-7, 10} + {2} or {-7, 9} + {2} are both non increasing.
LIS(4) is now 3, as we can extend {-7, 2} + {3} and this is the longest among other forms.
And so on until LIS(7). The answers are in LIS(5) or LIS(6), both with length 4.

There are clearly many overlapping sub-problems in LIS problem, but there are only N states, i.e. the LIS ending at index i, for all i $\in$ [0 ... N-1]. As we need to compute each state with an $O(n)$ loop, then this DP algorithm runs in $O(n^2)$. The LIS solution(s) can be reconstructed by following the arrows via some backtracking routine (scrutinize the arrows in Figure 3.9 for LIS(5) or LIS(6)).

Coin Change (CC) - The General Version

Problem[6]: 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

Recall that we can use greedy solution if the coin denominations are suitable – as in Example 1 above (See Section 3.3). But for general cases, we have to use DP – as in 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:
1. coin(0) = 0 // 0 coin to produce 0 cent
2. coin(< 0) = INVALID $\rightarrow$ (in practice, we just return a large positive value)
3. coin(value) = 1 + min(coin(value - coinValue[i])) $\forall i \in$ [0..N-1]
The answer is in coin(V).

[6]There are other variants of this problem, e.g. counting how many ways to do coin change and a variant where supply of coins are limited.

<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}

Both bottom up and top down DP happen to produce the same table...

Figure 3.10: Coin Change

In Figure 3.10, we see that:
coin(0) = 0, base case one.
coin(< 0) = ∞, base case two.
coin(1) = 1, from 1 + coin(1-1), as 1 + coin(1-5) is infeasible.
coin(2) = 2, from 1 + coin(2-1), as 1 + coin(2-5) is also infeasible.
... same thing for coin(3) and coin(4).
coin(5) = 1, from 1 + coin(5-5) = 1 coin, smaller than 1 + coin(5-1) = 5 coins.
And so on until coin(10). The answer is in coin(10), which is 2.

We can see that there are a lot of overlapping sub-problems in this Coin Change problem, but there are only $O(V)$ possible states! As we need to try $O(N)$ coins per state, the overall time complexity of this DP solution is $O(VN)$.

Maximum Sum

Abridged problem statement: Given n x n ($1 \leq n \leq 100$) array of integers, each ranging from [-127, 127], find a minimum sub-rectangle that have the maximum value.

Example: The 4 x 4 array ($n = 4$) below has 15 as the maximum sub-rectangle value.

```
 0  -2  -7   0            3 x 2 from lower left sub-rectangle
 9   2  -6   2     ==>    |  9  2 | = 9 + 2 - 4 + 1 - 1 + 8 = 15
-4   1  -4   1     ==>    | -4  1 |
-1   8   0  -2     ==>    | -1  8 |
```

Naïvely attacking this problem as shown below does not work as it is an 100^6 algorithm.

```
maxSubRect = -127*100*100; // lowest possible value for this problem
for (int i = 0; i < n; i++) for (int j = 0; j < n; j++) // start coord
  for  (int k = i; k < n; k++) for (int l = j; l < n; l++) { // end coord
    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); // keep largest so far
  }
```

Solution: There are several (well-known) DP solutions for *static* range problem. DP can work in this problem as computing a large sub-rectangle will definitely involves computing smaller sub-rectangles in it and such computation involves overlapping sub-rectangles!

One possible DP solution is to turn this n x n array into an n x n sum array where `arr[i][j]` no longer contains its own value, but the sum of all items within sub-rectangle (0, 0) to (i, j). This can easily be done on-the-fly when reading the input and still $O(n^2)$.

```
scanf("%d", &n);
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
```

This code turns input array (shown in the left) into sum array (shown in the right):

```
  0  -2  -7   0     ==>     0  -2  -9  -9
  9   2  -6   2     ==>     9   9  -4   2
 -4   1  -4   1     ==>     5   6 -11  -8
 -1   8   0  -2     ==>     4  13  -4  -3
```

Now, with this sum array, we can answer the sum of any sub-rectangle (i, j) to (k, l) in $O(1)$! Suppose we want to know the sum of (1, 2) to (3, 3). We can split the sum array into 4 sections and compute `arr[3][3] - arr[0][3] - arr[3][1] + arr[0][1] = -3 - 13 - (-9) + (-2) = -9`.

```
  0 [-2]| -9 [-9]
-----------------
  9   9 | -4   2
  5   6 |-11  -8
  4 [13]| -4 [-3]
```

With this $O(1)$ DP formulation, this problem can now be solved in 100^4.

```
maxSubRect = -127*100*100; // lowest possible value for this problem
for (int i = 0; i < n; i++) for (int j = 0; j < n; j++) // start coord
  for  (int k = i; k < n; k++) for (int l = j; l < n; l++) { // end coord
    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]; // inclusion-exclusion: O(1)
    maxSubRect = max(maxSubRect, subRect);
  }
```

Lesson: Not every range problems require Segment Tree as in Section 2.3.3! Problems where the input data is static like this usually is solvable with DP technique.

Exercise 1: The solution above runs in 100^4. There exist 100^3 solution. Can you figure out how to formulate this solution?

Exercise 2: What if the given static array is 1-D? Can you form a similar $O(1)$ DP solution to answer range sum query(i, j), i.e. `arr[i] + arr[i+1] + ... + arr[j]`?

Exercise 3: Use the solution from Exercise 2 to find maximum sum in 1-D array in $O(n^2)$. Can you further improve the solution to $O(n)$?

Exercise 4: Now, what if the query is range minimum query(i, j) on 1-D static array. The solution in Section 2.3.3 uses Segment Tree. Can you just utilize DP to solve the same problem – assuming that this time there is no update operation?

Remarks

There are other classical DP problems that we choose not to cover in this book such as Matrix Chain Multiplication [4], Optimal Binary Search Tree [14], 0-1 Knapsack [5, 14]. However, Floyd Warshall's will be discussed in Section 4.7; String Edit Distance, Longest Common Subsequence (LCS), plus other DP on Strings in Section 6.3.

3.4.3 Non Classical Examples

Although DP is one problem type that is most frequently appears 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 and build up our 'DP technique skills'. As you can see below, there are many interesting DP techniques from simple to intermediate. Try solving these problems at the corresponding online judges to further improve your DP skills. We have grouped and ordered the following DP problems from *our* perceived level of difficulty (easier to harder). There are DP trick lessons in each example. Enjoy :).

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 (coordinates within range $(0 \ldots l)$ 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: `coord` $= \{25, 50, 75\}$ (already sorted)

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

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

Solution: Use this Complete Search recurrences + top-down DP (memoization): cut(left, right)
1. If (left + 1 = right) where left and right are indices in array `coord`,
then cut(left, right) = 0 as we are left with 1 segment, there is no need to cut anymore.
2. Otherwise, try all possible cutting points and pick the minimum `cost` using code below:

```
// cost = initially infinity (a very large integer)
for (int i = left + 1; i < right; i++) // max 50 steps
  cost = min(cost, cut(left, i) + cut(i, right) + (coord[right] - coord[left]));
// after the loop above is done, we know that cut(left, right) = cost
```

Lesson: The Complete Search recurrences above only has 50×50 possible left/right indices configuration (states) and runs in just $50 \times 50 \times 50 = 125K$ operations! It is easy to convert this Complete Search / recursive backtracking recurrence into a top-down DP a.k.a memoization technique!

2. Forming Quiz Teams (UVa 10911)

For the abridged problem statement and full solution code, refer to the very first problem mentioned in Chapter 1. The grandiose name of this problem is "Minimum Weighted Perfect Matching on Small General Graph". In general, this problem is hard and the solution is Edmond's Matching algorithm (see [34]) which is not easy to code.

However, if the input size is small, up to $M \leq 20$, then the following DP + bitmasks technique can work. The idea is simple, as illustrated for $M = 6$: when nothing is matched yet, the state is `bit_mask=000000`. If item 0 and item 2 are matched, we can turn on bit 0 and bit 2 via these simple bit operations, i.e. `bit_mask | (1 << 0) | (1 << 2)`, thus the state becomes `bit_mask=000101`. Then, if from this state, item 1 and item 5 are matched, the state becomes `bit_mask=010111`. The perfect matching is obtained when the state is all '1's, in this case: `bit_mask=111111`.

Although there are many ways to arrive at a certain state, there are only $O(2^M)$ distinct states! For each state, we record the minimum weight of previous matchings that must be done in order to reach this state. As we want a perfect matching, then for a currently 'off' bit i, we must find the best other 'off' bit j from $[i{+}1 \ldots M{-}1]$ using one $O(M)$ loop. This check is again done using bit operation, i.e. `if (!(bit_mask & (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 + bitmasks approach is feasible. For more details, please study the code shown in Section 1.2.

Lesson: DP + bitmasks can solve small instances $M \leq 20$ of matching on general graph.

Exercise: The code in Section 1.2 says that "this 'break' is necessary. do you understand why?", with hint: "it helps reducing time complexity from $O((2N)^2 \times 2^{2N})$ to $O((2N) \times 2^{2N})$". Answer it!

3. Fishmonger (SPOJ 101)

Abridged problem statement: Given number of cities $3 \leq n \leq 50$, available time $1 \leq t \leq 1000$, and two $n \times n$ matrices (one gives a travel time and another gives tolls between two cities), choose a route from the first city 0 in such a way that one has to pay as little money for tolls as possible to arrive at the last city $n - 1$ within a certain time t. Output two information: the total tolls that is actually used and the actual traveling time.

Notice that there are *two* potentially conflicting requirements in this problem. Requirement one is to *minimize* tolls along the route. Requirement two is to *ensure* that we arrive in last city within allocated time, which may mean that we have to pay higher tolls in some part of the path. Requirement two is a *hard* constraint. We must satisfy it, otherwise we do not have a solution.

It should be quite clear after trying several test cases that greedy Single-Source Shortest Paths (SSSP) algorithm like Dijkstra's (Section 4.5) – on its pure form – will not work as picking the shortest travel time to ensure we use less than available time t may not lead to the smallest possible tolls. On the other hand, picking the cheapest tolls along the route may not ensure that we arrive within available time t. Both requirements cannot be made independent!

Solution: We can use the following Complete Search recurrences + top-down DP (memoization) `go(curCity, time_left)` that returns pairs of information (actual toll paid, actual time used):
1. `go(any curCity, < 0) = make_pair(INF, INF);` // Cannot go further if we run out of time.
2. `go(`$n-1$`, ≥ 0) = make_pair(0, 0);` // Arrive at last city, no need to pay/travel anymore.
3. In general case, `go(curCity, time_left)` is best described using the C++ code below:

```
pair<int, int> go(int curCity, int time_left) { // top-down DP, returns a pair
  if (time_left < 0) // invalid
    return make_pair(INF, INF); // a trick: return large value so that this state is not chosen
  if (curCity == n - 1) // at last city
    return make_pair(0, 0); // no need to pay toll, and time needed is 0
  if (memo[curCity][time_left].first != -1) // visited before
    return memo[curCity][time_left]; // simply return the answer

  pair<int, int> answer = make_pair(INF, INF);
  for (int neighbor = 0; neighbor < n; neighbor++) // try to go to another city
    if (curCity != neighbor) { // force us to always move
      pair<int, int> nextCity = go(neighbor,
        time_left - travelTime[curCity][neighbor]); // go to the other city
      // among neighboring cities, pick the one that has minimum cost
      if (nextCity.first + toll[curCity][neighbor] < answer.first) {
        answer.first = nextCity.first + toll[curCity][neighbor];
        answer.second = nextCity.second + travelTime[curCity][neighbor];
      }
    }
  return memo[curCity][time_left] = answer;
} // store the answer to memo table as well as returning the answer to the caller
```

Lesson: Some graph shortest (longest) path problems look solvable with classical (usually greedy) graph algorithms in Chapter 4, but in reality should be solved using DP techniques[7].

4. 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 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 maximum 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 3.11, the best descent path has a total of 8 acorns (dotted line).

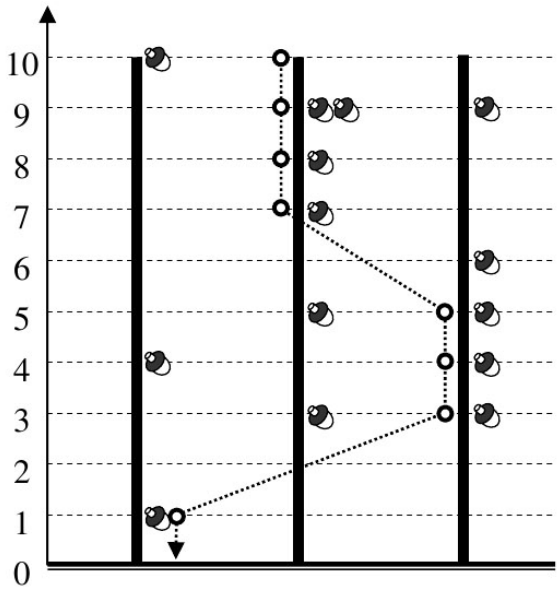


Figure 3.11: ACM ICPC Singapore 2007 - Jayjay the Flying Squirrel Collecting Acorns

[7]This problem can also be solved with a modified Dijkstra's.

Naïve DP Solution: Use a memo 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. This approach requires up to $2000 \times 2000 = 4M$ states and has time complexity $4M \times 2000 = 8B$ 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. 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 as follow:

```
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]); // solution will be here
```

Lesson: When naïve DP states are too large causing the overall DP time complexity not-doable, think of different ways *other than the obvious* to represent the possible states. Remember that no programming contest problem is unsolvable, the problem setter must have known a trick.

5. 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 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 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 max value is 100 + 100 + ... 100 (30 times) = 3000 and the min value is 100 - 100 - ... - 100 (29 times) = -2800 – this information needs to be stored, as we will see below.

The DP states 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.

But these two states are not unique yet. For example, this partial expression: '1-1+1-1...' has state 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 state idx=3, open=0 and sums to 0. But '1-(1+1)-1...' has the same state idx=3, open=0, *but* sums to -2. This DP state is *not yet* unique. So we need additional value to distinguish them, i.e. the value 'val', to make these states unique.

We can represent all possible states of this problem with `bool state[idx][open][val]`, a 3-D array. As 'val' ranges from -2800 to 3000 (5801 distinct values), the number of states is $30 \times 30 \times 5801 \approx 5M$ with only $O(1)$ processing per state – fast enough. The code is shown below:

```
void rec(int idx, int open, int val) {
  if (memo[idx][open][val + 3500]) // this state has been reached before
    return; // notice the + 3500 trick to convert negative indices to [0 .. 7000]
            // negative indices are not friendly for accessing a static array!
  memo[idx][open][val + 3500] = 1; // set this state to be reached

  if (idx == N) { // last number, current value is one of the possible
    used[val + 3500] = 1; // result of expression
    return;
  }

  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 if we
    rec(idx+1, open-1, nval); // already have some open brackets
  rec(idx+1, open, nval); // option 3: normal, do nothing
}

// Prepare a bool array 'used', initially set to all false,
//   then run this top-down DP.
// The solution is all the values in array 'used',
//   offset by -3500, that are flagged as true.
```

Lesson: 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.

6. Max Weighted Independent Set (on a Tree)

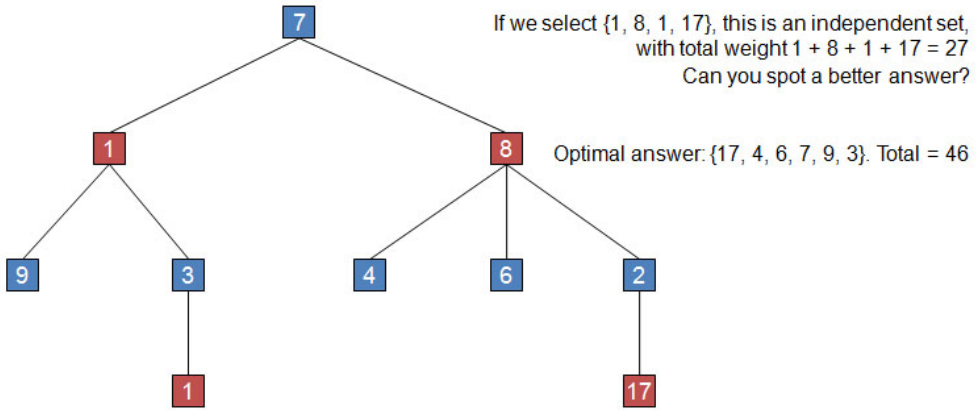


Figure 3.12: Max Weighted Independent Set (MWIS) on Tree

Abridged problem statement: Given a weighted graph G, find Max Weighted Independent Set (MWIS) on it. A subset of vertices of graph G is said to be Independent Set (IS)[8] if there is no edge in G between any two vertices in the IS. Our task is to select an IS with maximum weight of G. If graph G is a tree, this problem has efficient DP solution (see Figure 3.12).

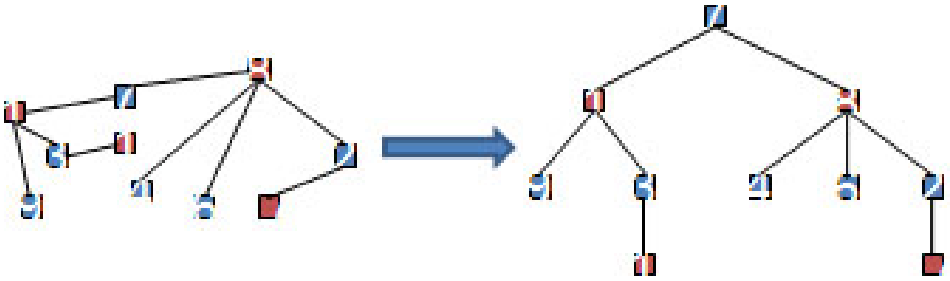


Figure 3.13: Root the Tree

Tip: For almost tree-related problems, we need to 'root the tree' first if it is not yet rooted. If the tree does not have a vertex dedicated as root, pick an arbitrary vertex as the root. By doing this, the subproblems w.r.t subtrees may appear, like in this MWIS problem on Tree (See Figure 3.13).

Once we have a rooted (sub)tree, we can formulate MWIS recurrences w.r.t their children. One formulation is: C(v, selected) = max weight of subtree rooted at 'v' if it is 'selected'.

Base cases (the leaves):
A. If leaf v is not selected, MWIS rooted at v is 0.
C(v, false) = 0 for all leaves v
B. If leaf v is selected, MWIS rooted at v is the weight of v.
C(v, true) = w(v) for all leaves v

Recurrence relations (the internal nodes):
A. If root v is taken, add weight of v but all children of v cannot be taken.
C(v, true) = $\sum_{\texttt{ch} \in \texttt{children(v)}}$ C(ch, false) + w(v)
B. If root v is not taken, children of v may or may not be taken (pick the maximum one).
C(v, false) = $\sum_{\texttt{ch} \in \texttt{children(v)}}$ max(C(ch, false), C(ch, true))

Answer will be in the root: max(C(root, false), C(root, true)). DP solution can be implemented either in a top-down or bottom-up fashion. As there are only $2 \times V$ states, the DP solution runs in $O(kV)$ where k is the average number of children of a vertex in the tree!

Lesson: Optimization problems on *tree* may be solved with DP techniques. The solution usually involves passing information to parent and getting information from the children of a rooted tree.

3.4.4 Remarks About Dynamic Programming in Programming Contests

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 'deciding problem'. But now, mastering DP is a *basic* requirement! You cannot do well in 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)

[8]FYI: The complement of Independent Set is Vertex Cover.

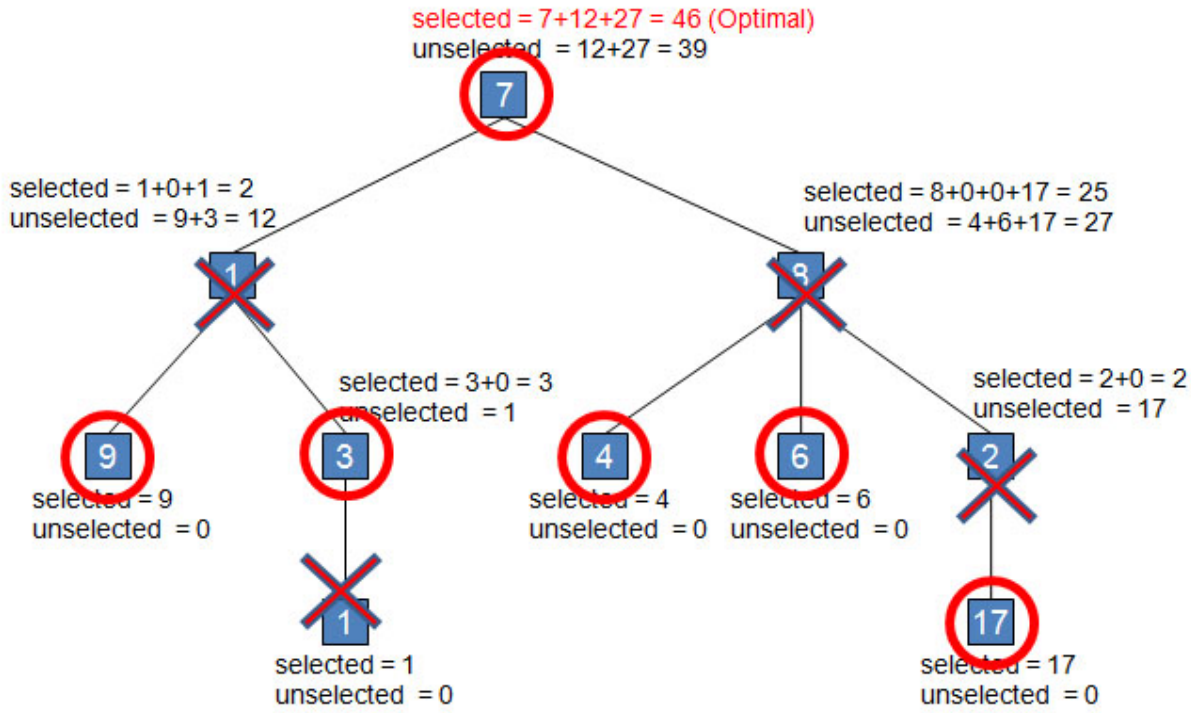


Figure 3.14: MWIS on Tree - The Solution

that can uniquely and efficiently represent sub-problems and then how to fill that table, either via top-down recursion or bottom-up loop.

We suggest that other than the examples shown in Section 3.4.3, contestants should learn newer forms of DP problems that keep appearing in recent programming contests, especially tricks on speeding up DP solution using 'quadrangle inequality', convex property, binary search, etc.

Programming Exercises for problems solvable using Dynamic Programming:

- Longest Increasing Subsequence (LIS) – Classical
  1. UVa 103 - Stacking Boxes (Longest Path in DAG ≈ LIS)
  2. UVa 111 - History Grading (straight-forward)
  3. UVa 231 - Testing the Catcher (straight-forward)
  4. UVa 481 - What Goes Up? (must use $O(n \log k)$ LIS)
  5. UVa 497 - Strategic Defense Initiative (solution must be printed)
  6. UVa 10051 - Tower of Cubes (can be modeled as LIS)
  7. UVa 10534 - Wavio Sequence (must use $O(n \log k)$ LIS twice)
  8. UVa 11790 - Murcia's Skyline (combination of classical LIS+LDS, weighted)
  9. UVa 11003 - Boxes
  10. UVa 11456 - Trainsorting (get max(LIS(i) + LDS(i) - 1), $\forall i \in [0 \ldots N\text{-}1]$)
  11. LA 2815 - Tiling Up Blocks (Kaohsiung03)
- Coin Change – Classical
  1. UVa 147 - Dollars (similar to UVa 357 and UVa 674)
  2. UVa 166 - Making Change
  3. UVa 357 - Let Me Count The Ways (a variant of the coin change problem)
  4. UVa 674 - Coin Change

5. UVa 10306 - e-Coins
6. UVa 10313 - Pay the Price
7. UVa 11137 - Ingenuous Cubrency (use long long)
8. UVa 11517 - Exact Change

- Maximum Sum
  1. UVa 108 - Maximum Sum (maximum 2-D sum, elaborated in this section)
  2. UVa 836 - Largest Submatrix (maximum 2-D sum)
  3. UVa 10074 - Take the Land (maximum 2-D sum)
  4. UVa 10667 - Largest Block (maximum 2-D sum)
  5. UVa 10827 - Maximum Sum on a Torus (maximum 2-D sum)
  6. UVa 507 - Jill Rides Again (maximum 1-D sum/maximum consecutive subsequence)
  7. UVa 10684 - The Jackpot (maximum 1-D sum/maximum consecutive subsequence)
- 0-1 Knapsack – Classical
  1. UVa 562 - Dividing Coins
  2. UVa 990 - Diving For Gold
  3. UVa 10130 - SuperSale
  4. LA 3619 - Sum of Different Primes (Yokohama06)
- String Edit (Alignment) Distance – Classical (see Section 6.3)
- Longest Common Subsequence – Classical (see Section 6.3)
- Non Classical (medium difficulty)
  1. UVa 116 - Unidirectional TSP (similar to UVa 10337)
  2. UVa 473 - Raucuous Rockers
  3. UVa 607 - Scheduling Lectures
  4. UVa 10003 - Cutting Sticks (discussed in this book)
  5. UVa 10337 - Flight Planner (DP solvable with Dijkstra)
  6. UVa 10891 - Game of Sum (2 dimensional states)
  7. UVa 11450 - Wedding Shopping (discussed in this book)
  8. LA 3404 - Atomic Car Race (Tokyo05)
- DP + Bitmasks
  1. UVa 10364 - Square (bitmask technique can be used)
  2. UVa 10651 - Pebble Solitaire
  3. UVa 10908 - Largest Square
  4. UVa 10911 - Forming Quiz Teams (elaborated in this section)
  5. LA 3136 - Fun Game (Beijing04)
  6. PKU 2441 - Arrange the Bulls
- DP on 'Graph Problem'
  1. UVa 590 - Always on the Run
  2. UVa 910 - TV Game (straightforward)
  3. UVa 10681 - Teobaldo's Trip
  4. UVa 10702 - Traveling Salesman
  5. LA 4201 - Switch Bulbs (Dhaka08)
  6. SPOJ 101 - Fishmonger

- DP with non-trivial states
  1. LA 4106 - ACORN (Singapore07) (DP with dimension reduction)
  2. LA 4143 - Free Parentheses (Jakarta08) (Problem set by Felix Halim)
  3. LA 4146 - ICPC Team Strategy (Jakarta08) (DP with 3 states)
  4. LA 4336 - Palindromic paths (Amritapuri08)
  5. LA 4337 - Pile it down (Amritapuri08)
  6. LA 4525 - Clues (Hsinchu09)
  7. LA 4526 - Inventory (Hsinchu09)
  8. LA 4643 - Twenty Questions (Tokyo09)
- DP on Tree
  1. UVa 10243 - Fire! Fire!! Fire!!! (Min Vertex Cover $\approx$ Max Independent Set on Tree)
  2. UVa 11307 - Alternative Arborescence (Min Chromatic Sum, 6 colors are sufficient)
  3. LA 3685 - Perfect Service (Kaohsiung06)
  4. LA 3794 - Party at Hali-Bula (Tehran06)
  5. LA 3797 - Bribing FIPA (Tehran06)
  6. LA 3902 - Network (Seoul07)
  7. LA 4141 - Disjoint Paths (Jakarta08)

3.5 Chapter Notes

The main source of the 'Complete Search' material in this chapter is the USACO training gateway [18].

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, e.g. **Branch & Bound** and **A* Search** (both are not discussed yet in this book). We believe the term 'clever Brute-Force' is a bit self-contradicting.

Divide and Conquer paradigm is usually used in form of its popular algorithms: binary search (bisection method), merge/quick/heap sort, and data structures: binary search tree, heap, segment tree, etc. We will see more Divide and Conquer later in Computational Geometry (Section 7.5).

Basic Greedy and Dynamic Programming (DP) techniques techniques are always included in popular algorithm textbooks, e.g. Introduction to Algorithms [4], Algorithm Design [13], Algorithm [5], Programming Challenges [23], The Art of Algorithms and Programming Contests [16], Art of Programming Contest [1]. 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 [25] and recent programming contests, e.g. ACM ICPC Regional Kaohsiung 2006, Singapore 2007, Jakarta 2008, Central European Olympiad in Informatics 2009, etc.

However, for some real-life problems, especially those that are classified as NP-Complete [4], many of the approaches discussed so far will not work. For example, 0-1 Knapsack Problem which has $O(nW)$ DP complexity is too slow if W is big. For this, people use heuristics or local search: Beam Search, Tabu Search [10], Genetic Algorithm, Ants Colony Optimization, etc.

There are approximately **109 programming exercises** discussed in this chapter.

Chapter 4

Graph

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

Many real-life problems can be classified as graph problems. Some have efficient solutions. Some do not have them yet. In this chapter, we learn various graph problems with known efficient solutions, ranging from basic traversal, minimum spanning trees, shortest paths, and network flow algorithms.

4.1 Overview and Motivation

In this chapter, we discuss graph problems that are commonly appear in programming contests, the algorithms to solve them, and the practical implementations of these algorithms. The issue on how to store graph information has been discussed earlier in Section 2.3.1.

We assume that the readers are familiar with the following 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, Tree/Forest, Complete Graph, Directed Acyclic Graph, Bipartite Graph, Euler/Hamiltonian Path/Cycle. If you encounter any unknown terms. Please go to Wikipedia [46] and search for that particular term.

Table 4.1 summarizes our research so far on graph problems in recent ACM ICPC Asia regional contests and its ICPC Live Archive [11] IDs. Although there are so many graph problems (many are discussed in this chapter), they only appear either once or twice in a problem set. The question is: "Which ones that we have to learn?" If you want to do well in contests, you have no choice but to study all these materials.

4.2 Depth First Search

Basic Form and Application

Depth First Search - abbreviated as DFS - is a simple algorithm to traverse a graph. With the help of a global array `int dfs_num[V]` which is initially set to 'unvisited' (a constant value

LA	Problem Name	Source	Graph Problem (Algorithm)
2817	The Suspects	Kaohsiung03	Connected Component
2818	Geodetic Set Problem	Kaohsiung03	APSP (Floyd Warshall)
3133	Finding Nemo	Beijing04	SSSP (Dijkstra on Grid)
3138	Color a Tree	Beijing04	DFS
3171	Oreon	Manila06	MST (Kruskal)
3290	Invite Your Friends	Dhaka05	SSSP Un+Weighted (BFS + Dijkstra)
3294	Ultimate Bamboo Eater	Dhaka05	Longest Path in DAG (Toposort + DP + DS)
3678	Bug Sensor Problem	Kaohsiung06	MST (Kruskal)
4099	Sub-dictionary	Iran07	SCC
4109	USHER	Singapore07	SSSP (Floyd Warshall, small graph)
4110	RACING	Singapore07	'Maximum' Spanning Tree (Kruskal)
4138	Anti Brute Force Lock	Jakarta08	MST (Kruskal)
4271	Necklace	Hefei08	Flow on Graph
4272	Polynomial-time Red...	Hefei08	SCC
4407	Gun Fight	KLumpur08	Bipartite Matching ($\approx$ Max Flow)
4408	Unlock the Lock	KLumpur08	SSSP on unweighted graph (BFS)
4524	Interstar Transport	Hsinchu09	APSP (Floyd Warshall)
4637	Repeated Substitution ...	Japan09	SSSP (BFS)
4645	Infected Land	Japan09	SSSP (BFS)

Table 4.1: Some Graph Problems in Recent ACM ICPC Asia Regional

`DFS_WHITE = -1`), DFS starts from a vertex u, mark u as 'visited' (set `dfs_num[u]` to `DFS_BLACK = 1`), and then for each 'unvisited' neighbor v of u (i.e. edge $u - v$ exist in the graph), recursively visit v. The snippet of DFS code is shown below:

```
typedef pair<int, int> ii; // we will frequently use these two data type shortcuts
typedef vector<ii> vii;
#define TRvii(c, it) \ // all sample codes involving TRvii use this macro
  for (vii::iterator it = (c).begin(); it != (c).end(); it++)

void dfs(int u) { // DFS for normal usage
  printf(" %d", u); dfs_num[u] = DFS_BLACK; // this vertex is visited, mark it
  TRvii (AdjList[u], v) // try all neighbors v of vertex u
    if (dfs_num[v->first] == DFS_WHITE) // avoid cycle
      dfs(v->first); // v is a (neighbor, weight) pair
}
```

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.

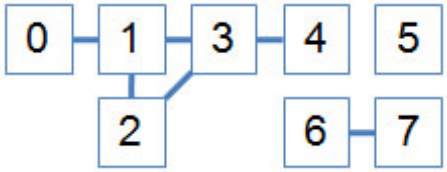


Figure 4.1: Sample graph for the early part of this section

On 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 branches. Note that this sequence of visitation depends very much on the way we order 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. 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, and 7 in Figure 4.1 are currently unvisited by calling `dfs(0)`.

The DFS code shown here is very similar to the recursive backtracking code shown earlier in Section 3.1. 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 just whether we flag visited vertices. DFS does. Backtracking does not. By not revisiting vertices, DFS runs in $O(V + E)$, but the time complexity of backtracking goes up exponentially.

```
void backtracking(state) {
  if (hit end state or invalid state) // invalid state includes states that cause cycling
    return; // we need terminating/pruning condition
  for each neighbor of this state // regardless it has been visited or not
    backtracking(neighbor);
}
```

Other Applications

DFS is not only useful for traversing a graph. It can be used to solve many other graph problems.

Finding Connected Components in Undirected Graph

The fact that one single call of `dfs(u)` will only visit vertices that are actually connected to u can be utilized to find (and to count) the connected components of an undirected graph (see further below for a similar problem on directed graph). We can simply use the following code to restart DFS from one of the remaining unvisited vertices to find the next connected component (until all are visited):

```
#define REP(i, a, b) \ // all sample codes involving REP use this macro
  for (int i = int(a); i <= int(b); i++)

// inside int main()
numComponent = 0;
memset(dfs_num, DFS_WHITE, sizeof dfs_num);
REP (i, 0, V - 1)
  if (dfs_num[i] == DFS_WHITE) {
    printf("Component %d, visit:", ++numComponent);
    dfs(i);
    printf("\n");
  }
printf("There are %d connected components\n", numComponent);

// For the sample graph, the output is like this:
// Component 1, visit: 0 1 2 3 4
// Component 2, visit: 5
// Component 3, visit: 6 7
```

Exercise: We can also use Union-Find Disjoint Sets to solve this graph problem. How?

Flood Fill - Labeling the Connected Components

DFS can be used for other purpose than just finding (and counting) the connected components. Here, we show how a simple tweak of `dfs(u)` can be used to *label* the components. Typically, we 'label' (or 'color' the component by using its component number. This variant is more famously known as 'flood fill'.

```
void floodfill(int u, int color) {
  dfs_num[u] = color; // not just a generic DFS_BLACK
  TRvii (AdjList[u], v) // try all neighbors v of vertex u, note that we use our TRvii macro again
    if (dfs_num[v->first] == DFS_WHITE) // avoid cycle
      floodfill(v->first, color); // v is an (neighbor, weight) pair
}

// inside int main()
numComponent = 0;
memset(dfs_num, DFS_WHITE, sizeof dfs_num);
REP (i, 0, V - 1) // for each vertex u in [0..V-1], note that we use our REP macro again
  if (dfs_num[i] == DFS_WHITE) // if not visited
    floodfill(i, ++numComponent);
REP (i, 0, V - 1)
  printf("Vertex %d has color %d\n", i, dfs_num[i]);

// For the sample graph, the output is like this:
// Vertex 0-1-2-3-4 have color 1
// Vertex 5 has color 2
// Vertex 6-7 has color 3
```

Exercise: Flood Fill is more commonly performed on 2-D grid (implicit graph). Try to solve UVa 352, 469, 572, etc.

Graph Edges Property Check via DFS Spanning Tree

Running DFS on a connected component of a graph will form a DFS *spanning tree* (or *spanning forest* if the graph has more than one component and DFS is run on each component). With one more vertex state: `DFS_GRAY = 2` (visited but not yet completed) on top of `DFS_WHITE` (unvisited) and `DFS_BLACK` (visited and completed), we can use this DFS spanning tree (or forest) to classify graph edges into four 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 cycle, i.e. from vertex with `DFS_GRAY` to vertex with `DFS_GRAY` too.
   Note that usually we do not count bi-directional edges as having '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 problem.

Figure 4.2 shows an animation (from top left to bottom right) of calling `dfs(0)`, 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 edge $(1 \rightarrow 0)$ is just a bi-directional edge. The code for this DFS variant is shown below.

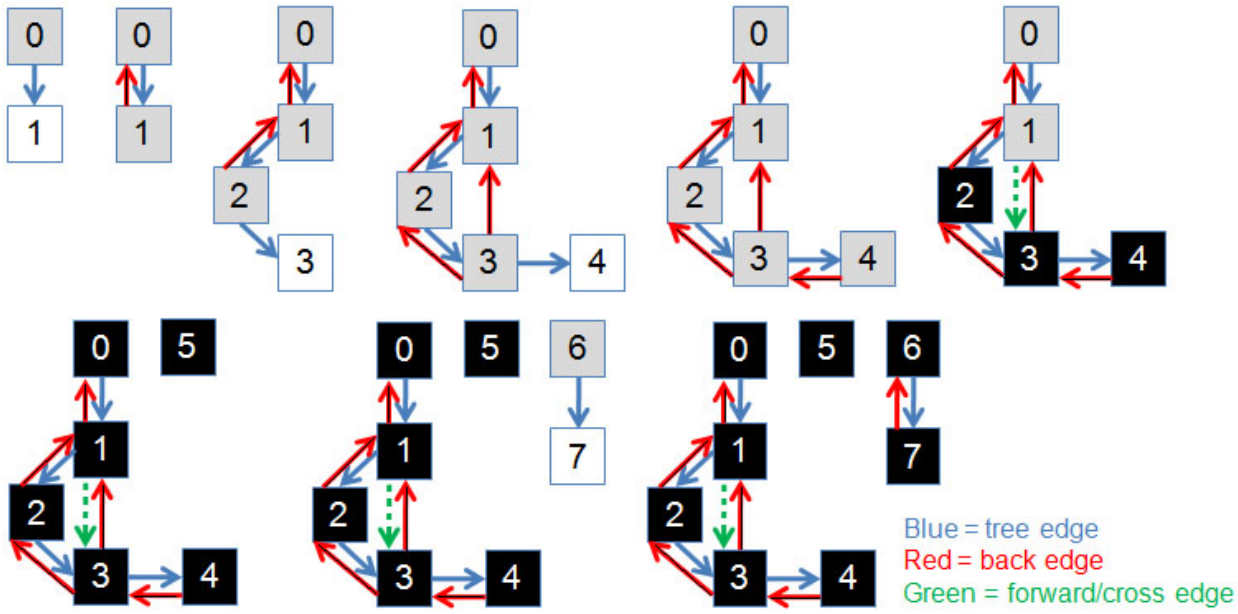


Figure 4.2: Animation of DFS

```
void graphCheck(int u) { // DFS for checking graph edge properties...
  dfs_num[u] = DFS_GRAY; // color this as GRAY (temporary)
  TRvii (AdjList[u], v) { // traverse this AdjList
    if (dfs_num[v->first] == DFS_WHITE) { // GRAY to WHITE
      // printf("  Tree Edge (%d, %d)\n", u, v->first);
      dfs_parent[v->first] = u; // parent of this children is me
      graphCheck(v->first);
    }
    else if (dfs_num[v->first] == DFS_GRAY) { // GRAY to GRAY
      if (v->first == dfs_parent[u])
        printf(" Bidirectional (%d, %d) - (%d, %d)\n", u, v->first, v->first, u);
      else
        printf(" Back Edge (%d, %d) (Cycle)\n", u, v->first);
    }
    else if (dfs_num[v->first] == DFS_BLACK) // GRAY to BLACK
      printf(" Forward/Cross Edge (%d, %d)\n", u, v->first);
  }
  dfs_num[u] = DFS_BLACK; // now color this as BLACK (DONE)
}
```

Finding Articulation Points and Bridges

Motivating problem: Given a road map (undirected graph) with cost 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 disconnects G. A graph without any articulation points 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 (can be tweaked to find bridges too):

1. Run $O(V + E)$ DFS to count number of connected components of the original graph
2. For all vertex $v \in V$ // $O(V)$
   (a) Cut (remove) vertex v and its incident edges
   (b) Run $O(V + E)$ DFS to check if number of connected components increase
   (c) If yes, v is an articulation point/cut vertex; Restore v and its incident edges

This naïve algorithm calls DFS $O(V)$ 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 Hopcroft and Robert Endre Tarjan (see problem 22.2 in [4]), 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)` stores the iteration counter when the vertex u is visited *for the first time* and 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 (some back edges exist). Note that we do not update `dfs_low(u)` with back edge (u, v) if v is a direct parent of u.

See Figure 4.3 for clarity. In these two sample graphs, we run `articulationPointAndBridge(0)`. Suppose in the graph in Figure 4.3 – 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`. Since there is no back edge in this graph, all `dfs_low = dfs_num`.

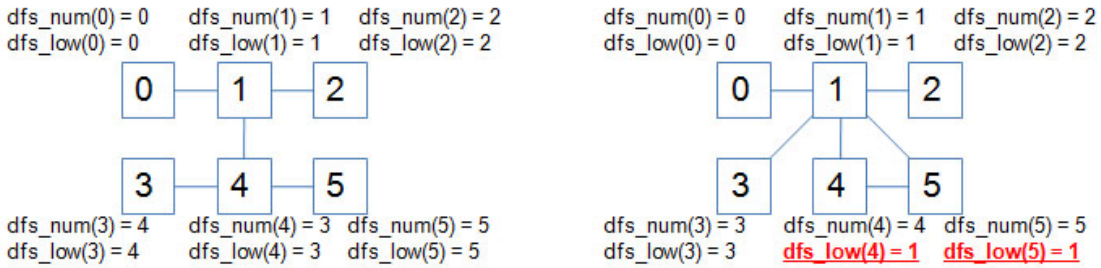


Figure 4.3: Introducing two more DFS attributes: `dfs_number` and `dfs_low`

In the graph in Figure 4.3 – 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). 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, v is a neighbor of u, 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 parent of u in the DFS spanning tree). Thus, to reach that parent of u from v, one must pass through vertex u. This implies that removing the vertex u will disconnect the graph.

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).

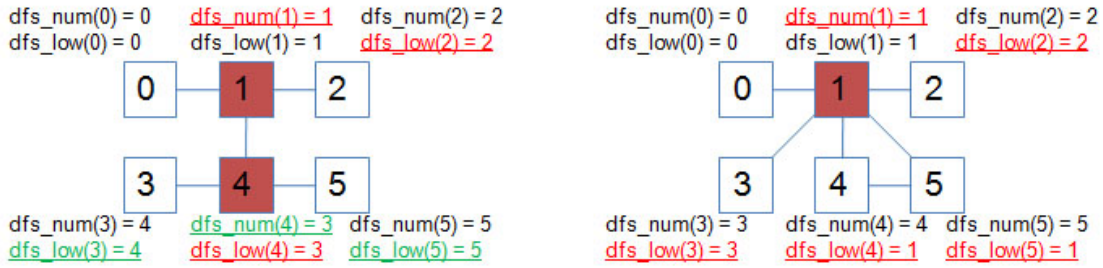


Figure 4.4: Finding articulation points with `dfs_num` and `dfs_low`

See Figure 4.4 for more details. On the graph in Figure 4.4 – 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 4.4 – right side, 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 4.5, 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. 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`.

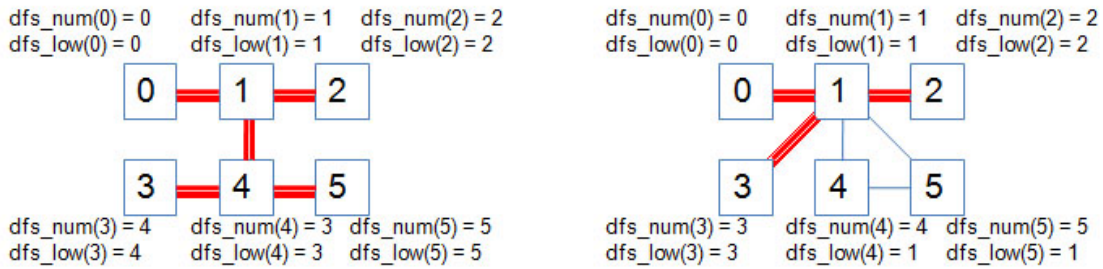


Figure 4.5: Finding bridges, also with `dfs_num` and `dfs_low`

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]
  TRvii (AdjList[u], v)
    if (dfs_num[v->first] == DFS_WHITE) { // a tree edge
      dfs_parent[v->first] = u; // parent of this children is me
      if (u == dfsRoot) // special case
        rootChildren++; // 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;
memset(dfs_num, DFS_WHITE, sizeof dfs_num);
printf("Bridges:\n");
REP (i, 0, V - 1)
  if (dfs_num[i] == DFS_WHITE) {
    dfsRoot = i; rootChildren = 0;
    articulationPointAndBridge(i);
    articulation_vertex[dfsRoot] = (rootChildren > 1); // special case
  }

printf("Articulation Points:\n");
REP (i, 0, V - 1)
  if (articulation_vertex[i])
    printf(" Vertex %d\n", i);
```

Finding Strongly Connected Components in 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 of an undirected graph. In Figure 4.6, we have a similar graph to the graph in Figure 4.1, but now the edges are directed. We can see that although the graph in Figure 4.6 looks like 'connected' into one component, it is actually 'not strong'. In directed graph, we are more interested with the notion of 'Strongly Connected Component (SCC)', i.e. 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 4.6, as highlighted: {0}, {1, 3, 2}, and {4, 5, 7, 6}.

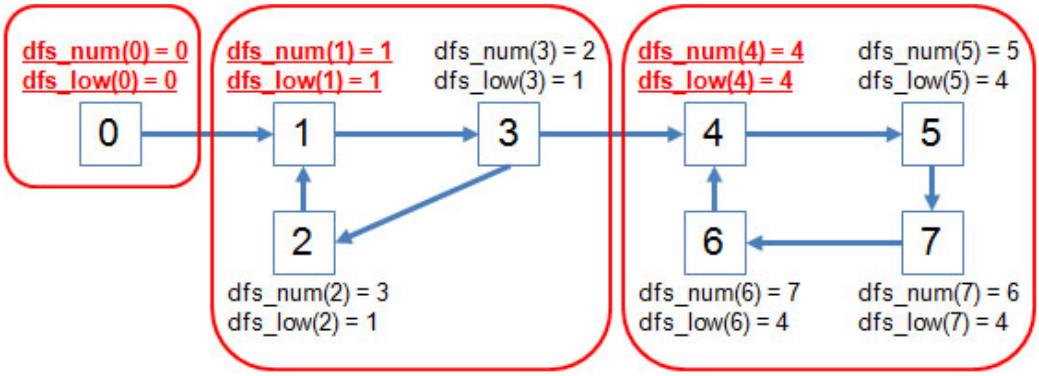


Figure 4.6: An example of directed graph and its Strongly Connected Components (SCC)

There are at least two known algorithms to find SCCs: Kosaraju's – explained in [4] and Tarjan's algorithm [45]. 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 the subtrees of the DFS spanning tree and the roots of the subtrees are also the roots of the SCCs. To determine whether a vertex u is the root of an SCC, Tarjan's SCC algorithm uses `dfs_num` and `dfs_low`, i.e. by checking if `dfs_low(u) = dfs_num(u)`. The visited vertices are pushed into a stack according to its `dfs_num`. When DFS returns from a subtree, the vertices are popped from the stack. If the vertex is the root of an SCC, then that vertex and all of the vertices popped before it forms that SCC. The code is shown below:

```
stack<int> dfs_scc; // additional information for SCC
set<int> in_stack; // for dfs_low update check

void tarjanSCC(int u) {
  dfs_low[u] = dfs_num[u] = dfsNumberCounter++; // dfs_low[u] <= dfs_num[u]
  dfs_scc.push(u); in_stack.insert(u); // stores u based on order of visitation
  TRvii (AdjList[u], v) {
    if (dfs_num[v->first] == DFS_WHITE) // a tree edge
      tarjanSCC(v->first);
    if (in_stack.find(v->first) != in_stack.end()) // condition for update
      dfs_low[u] = min(dfs_low[u], dfs_low[v->first]); // update dfs_low[u]
  }
  if (dfs_low[u] == dfs_num[u]) { // if this is a root of SCC
    printf("SCC: ");
    while (!dfs_scc.empty() && dfs_scc.top() != u) {
      printf("%d ", dfs_scc.top()); in_stack.erase(dfs_scc.top()); dfs_scc.pop();
    }
    printf("%d\n", dfs_scc.top()); in_stack.erase(dfs_scc.top()); dfs_scc.pop();
} }
```

Exercise: This implementation can be improved by about $O(\log V)$ factor by using other data structure other than `set<int> in_stack`. How?

Topological Sort (on 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. There are several ways to implement a Topological Sort algorithm. The simplest is to slightly modify the simplest DFS implementation in this section.

```
void topoVisit(int u) {
  dfs_num[u] = DFS_BLACK;
  TRvii (AdjList[u], v)
    if (dfs_num[v->first] == DFS_WHITE)
      topoVisit(v->first);
  topologicalSort.push_back(u); // this is the only change
}

// inside int main()
topologicalSort.clear(); // this global vector stores topological sort in reverse order
memset(dfs_num, DFS_WHITE, sizeof dfs_num);
REP (i, 0, V - 1)
  if (dfs_num[i] == DFS_WHITE)
    topoVisit(i);
reverse(topologicalSort.begin(), topologicalSort.end());
REP (i, 0, topologicalSort.size() - 1)
  printf("%d\n", topologicalSort[i]);
```

In `topoVisit(u)`, we append u to the list vertices explored only after visiting all subtrees below u. As `vector` only support *efficient insertion* from back, we work around this issue by simply reversing the print order in the output phase. This simple algorithm for finding (a valid) topological sort is again due to Tarjan. It again runs in $O(V + E)$ as with DFS.

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

Programming Exercises for Depth First Search (DFS):

- Finding Connected Components / Flood Fill
  1. UVa 260 - Il Gioco dell'X
  2. UVa 352 - Seasonal War (Flood Fill)
  3. UVa 459 - Graph Connectivity (also solvable with Union-Find Disjoint Sets)
  4. UVa 469 - Wetlands of Florida (Flood Fill)
  5. UVa 572 - Oil Deposits (Flood Fill)
  6. UVa 657 - The Die is Cast (Flood Fill)
  7. UVa 782 - Countour Painting (Flood Fill)
  8. UVa 784 - Maze Exploration (Flood Fill)
  9. UVa 785 - Grid Colouring (Flood Fill)
  10. UVa 852 - Deciding victory in Go (Flood Fill)
  11. UVa 10336 - Rank the Languages (Flood Fill)
  12. UVa 10926 - How Many Dependencies?
  13. UVa 10946 - You want what filled? (Flood Fill)
  14. UVa 11110 - Equidivisions (Flood Fill)
  15. UVa 11518 - Dominos 2 (Flood Fill)
  16. UVa 11749 - Poor Trade Advisor (also solvable with Union-Find Disjoint Sets)
- Finding Articulation Points / Bridges
  1. UVa 315 - Network (Articulation Points)
  2. UVa 610 - Street Directions (Bridges)
  3. UVa 796 - Critical Links (Bridges)
  4. UVa 10199 - Tourist Guide (Articulation Points)
- Finding Strongly Connected Components
  1. UVa 10731 - Test
  2. UVa 11504 - Dominos
  3. UVa 11709 - Trust Groups
  4. UVa 11770 - Lighting Away
- Topological Sort (also see shortest/longest paths problems in DAG in Section 4.9.2)
  1. UVa 124 - Following Orders
  2. UVa 200 - Rare Order
  3. UVa 872 - Ordering
  4. UVa 10305 - Ordering Tasks

4.3 Breadth First Search

Basic Form and Application

Breadth First Search - abbreviated as BFS - is another form of graph traversal algorithm. BFS starts with the insertion of initial source vertex s into a queue, then processes the queue as follows:

take out the front most vertex u from the queue and enqueue each unvisited neighbors of u. With the help of the queue, BFS will visit vertex s and all vertices in the connected component that contains s layer by layer. This is why the name is *breadth-first*. BFS algorithm also runs in $O(V+E)$ on a graph represented using an Adjacency List.

Implementing BFS is easy if we utilize C++ STL libraries. 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 source vertex. This feature is important as it can be used to solve special case of Single-Source Shortest Paths problem (discussed below).

```
queue<int> q; map<int, int> dist;
q.push(s); dist[s] = 0; // start from source

while (!q.empty()) {
  int u = q.front(); q.pop(); // queue: layer by layer!
  printf("Visit %d, Layer %d\n", u, dist[u]);
  TRvii (AdjList[u], v) // for each neighbours of u
    if (!dist.count(v->first)) { // dist.find(v) != dist.end() also works
      dist[v->first] = dist[u] + 1; // if v not visited before + reachable from u
      q.push(v->first); // enqueue v for next steps
}   }
```

Exercise: This implementation uses `map<STATE-TYPE, int> dist` to store distance information. This may be useful if `STATE-TYPE` is not integer, e.g. a `pair<int, int>` of (row, col) coordinate. However, this trick adds a $\log V$ factor to the $O(V + E)$ BFS complexity. Please, rewrite this implementation to use `vector<int> dist` instead!

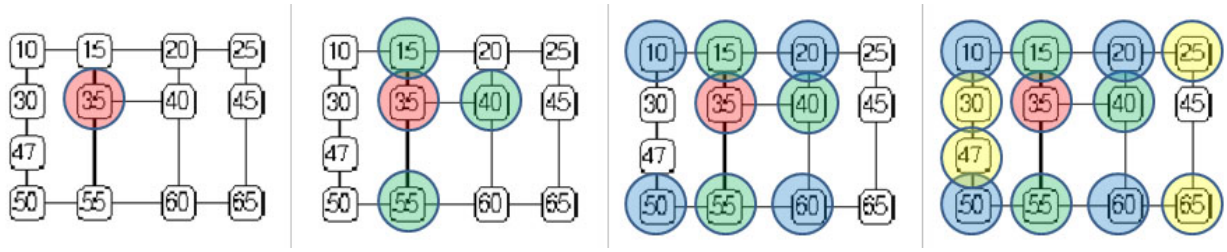


Figure 4.7: Animation of BFS (from UVa 336 [17])

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 4.7, we will visit the vertices in the following order:

```
Layer 0: >35< (source)
Layer 1: 15, 55, 40
Layer 2: 10, 20, 50, 60
Layer 3: >30<, 25, 47, 65
Layer 4: 45
// Three layers from '35' to '30' implies that the shortest path between them
// on this unweighted graph is 3 distance units.
```

Other Applications

Single-Source Shortest Paths (SSSP) on Unweighted Graph

The fact that BFS visits vertices of a graph layer by layer from a source vertex turns BFS as a good solver for Single-Source Shortest Paths (SSSP) problem on unweighted graph. 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 4.7, 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. Reconstructing the shortest path: $35 \rightarrow 15 \rightarrow 10 \rightarrow 30$ is easy if we store the BFS spanning tree, i.e. vertex '30' remembers '10' as its parent, vertex '10' remembers '15', vertex '15' remembers '35' (the source).

Which graph traversal algorithm to choose? Table 4.2 can be helpful.

	$O(V+E)$ **DFS**	$O(V+E)$ **BFS**
Pro	Uses less memory	Can solve SSSP on unweighted graphs
Cons	Cannot solve SSSP on unweighted graphs	Uses more memory
Code	Slightly easier to code	Slightly longer to code

Table 4.2: Graph Traversal Algorithm Decision Table

Programming Exercises for Breadth First Search (BFS):

- Single-Source Shortest Paths on Unweighted Graph
  1. UVa 336 - A Node Too Far
  2. UVa 383 - Shipping Routes
  3. UVa 417 - Word Index
  4. UVa 429 - Word Transformation
  5. UVa 439 - Knight Moves
  6. UVa 532 - Dungeon Master (3-D BFS)
  7. UVa 567 - Risk
  8. UVa 627 - The Net (must print the path)
  9. UVa 762 - We Ship Cheap
  10. UVa 924 - Spreading the News
  11. UVa 928 - Eternal Truths
  12. UVa 10009 - All Roads Lead Where?
  13. UVa 10044 - Erdos numbers (parsing part is troublesome)
  14. UVa 10067 - Playing with Wheels (implicit graph in problem statement)
  15. UVa 10102 - The Path in the Colored Field
  16. UVa 10150 - Doublets (BFS state is string!)
  17. UVa 10422 - Knights in FEN
  18. UVa 10610 - Gopher and Hawks
  19. UVa 10653 - Bombs! NO they are Mines!! (BFS implementation must be efficient)
  20. UVa 10959 - The Party, Part I
  21. UVa 11049 - Basic Wall Maze (some restricted moves + print path)
  22. UVa 11352 - Crazy King
  23. UVa 11513 - 9 Puzzle (BFS from goal state + efficient data structure)
  24. UVa 11545 - Avoiding Jungle in the Dark
  25. UVa 11730 - Number Transformation (need prime factoring, see Section 5.3.1)
  26. UVa 11792 - Krochanska is Here! (be careful with the definition of 'important station')
  27. LA 3290 - Invite Your Friends (plus Dijkstra's)

28. LA 4408 - Unlock the Lock
29. LA 4637 - Repeated Substitution with Sed
30. LA 4645 - Infected Land

- Variants
  1. UVa 10004 - Bicoloring (Bipartite Graph check)
  2. UVa 11080 - Place the Guards (Bipartite Graph check, some tricky cases)
  3. UVa 11101 - Mall Mania (Multi-Sources BFS from mall1, get min at border of mall2)
  4. UVa 11624 - Fire! (Multi-Sources BFS)

4.4 Kruskal's

Basic Form and Application

Motivating problem: Given a connected, undirected, and weighted graph G (see the leftmost graph in Figure 4.8), 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 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 4.8, middle and right sides. One of them is the required solution that satisfies the minimal weight criteria.

This problem is called the Minimum Spanning Tree (MST) problem and has many practical applications, as we will see later in this section.

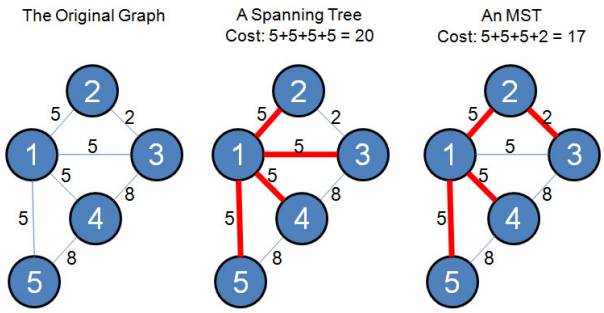


Figure 4.8: Example of a Minimum Spanning Tree (MST) Problem (from UVa 908 [17])

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 [4, 21, 14, 23, 16, 1, 13, 5]. For programming contests, we opt Kruskal's as it's implementation is very easy with help of 2 data structures.

Joseph Bernard *Kruskal* Jr.'s algorithm first sort E edges based on non decreasing weight in $O(E \log E)$. This can be easily done using `priority_queue` (or alternatively, use `vector` & `sort`). Then, it *greedily* tries to add $O(E)$ edges with minimum costs to the solution as long as such addition does not form a cycle. This cycle check can be done easily using Union-Find Disjoint Sets. The code is short and in overall runs in $O(E \log E)$.

```
#typedef pair<int, int> ii; // we use ii as a shortcut of integer pair data type

priority_queue< pair<int, ii> > EdgeList; // sort by edge weight O(E log E)
// PQ default: sort descending. To sort ascending, we can use <(negative) weight(i, j), <i, j> >
```

```
// for each edge with (i, j, weight) format
//   EdgeList.push(make_pair(-weight, make_pair(i, j)));
// alternative implementation: use STL vector and algorithm:sort

mst_cost = 0; initSet(V); // all V are disjoint sets initially, see Section 2.3.2
while (!EdgeList.empty()) { // while there exist more edges, O(E)
  pair<int, ii> front = EdgeList.top(); EdgeList.pop();
  if (!isSameSet(front.second.first, front.second.second)) { // if no cycle
    mst_cost += (-front.first); // add (negated) -weight of e to MST
    unionSet(front.second.first, front.second.second); // link these two vertices
} }
// note that the number of disjoint sets must eventually be one!
// otherwise, no MST has been formed...
```

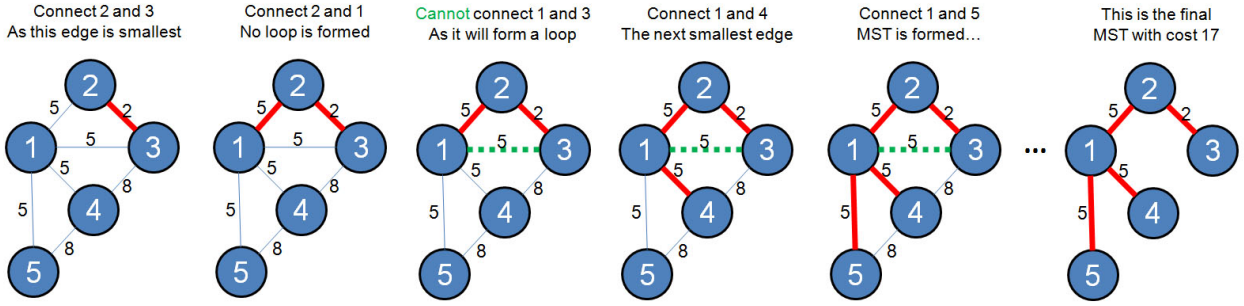


Figure 4.9: Kruskal's Algorithm for MST Problem (from UVa 908 [17])

Exercise: The implementation shown here only stop when EdgeList is empty. For some cases, we can stop Kruskal's algorithm earlier. When and how to modify the code to handle this?
Figure 4.9 shows the execution of Kruskal's algorithm on the graph shown in Figure 4.8, leftmost[1].

Other Applications

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

'Maximum' Spanning Tree

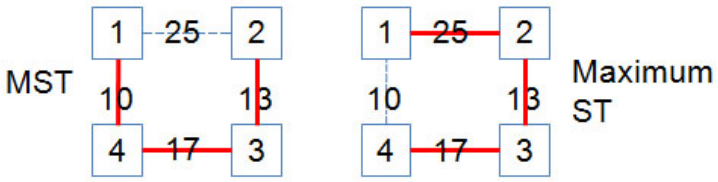


Figure 4.10: 'Maximum' Spanning Tree Problem

This is a simple variant where we want the maximum, instead of the minimum ST. In Figure 4.10, we see a comparison between MST and Maximum ST. Solution: sort edges in non increasing order.

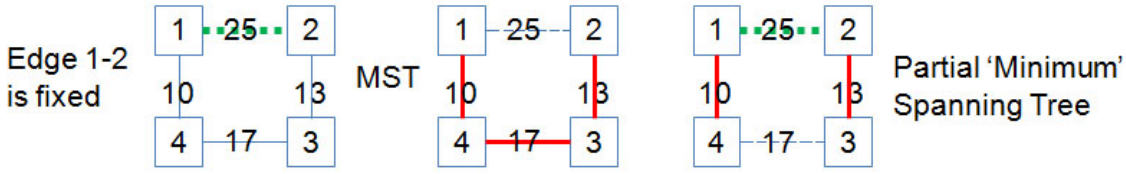


Figure 4.11: Partial 'Minimum' Spanning Tree Problem

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 perhaps no longer minimum overall. That's why we put the term 'Minimum' in quotes. In Figure 4.11, we see an example when one edge 1-2 is already fixed (left). The actual MST is 10+13+17 = 40 which omits the edge 1-2 (middle). However, the solution for this problem must be (25)+10+13 = 48 which uses the edge 1-2 (right).

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'

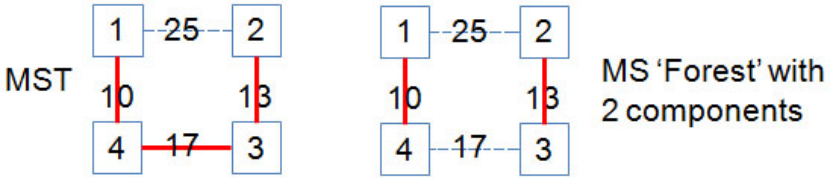


Figure 4.12: Minimum Spanning 'Forest' Problem

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, left, we observe that the MST for this graph is 10+13+17 = 40. But if we want a spanning forest with 2 components, then the solution is just 10+13 = 23 on Figure 4.12, right.

To get minimum spanning forest is simple. Run Kruskal's algorithm as per normal, but as soon as the number of connected component equals to the desired pre-determined number, we can terminate the algorithm.

Second Best Spanning Tree

Sometimes, we are interested to have a backup plan. 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 MST and another chord[2] edge is added to MST. In this example: the edge 4-5 is taken out and the edge 2-5 is added in.

[1]Note that the solution for this problem is definitely a tree, i.e. no cycles in the solution!
[2]A chord edge is defined as an edge in graph G that is not selected in the MST of G.

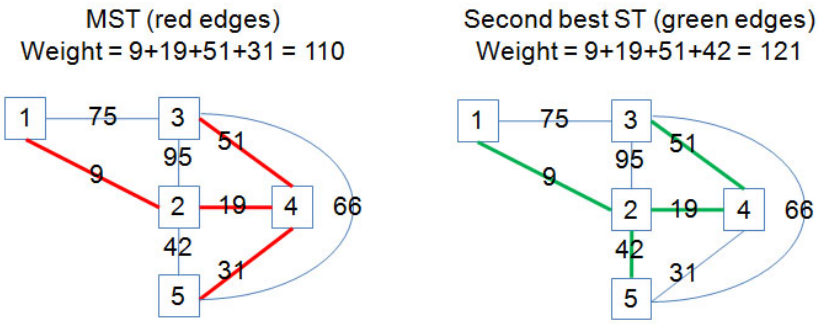


Figure 4.13: Second Best Spanning Tree (from UVa 10600 [17])

A simple solution is to first sort the edges in $O(E \log E)$, then find the MST using Kruskal's in $O(E)$. Then, for each edge in the MST, make its weight to be `INF` (infinite) to 'delete' it – in practice, this is just a very big number. Then try to find the second best ST in $O(VE)$ as there are $E = V - 1$ edges in the MST that we have to try. Figure 4.14 shows this algorithm on the given graph. Remember that both MST and second best ST are spanning tree, i.e. they are connected! In overall, this algorithm runs in $O(E \log E + E + VE)$.

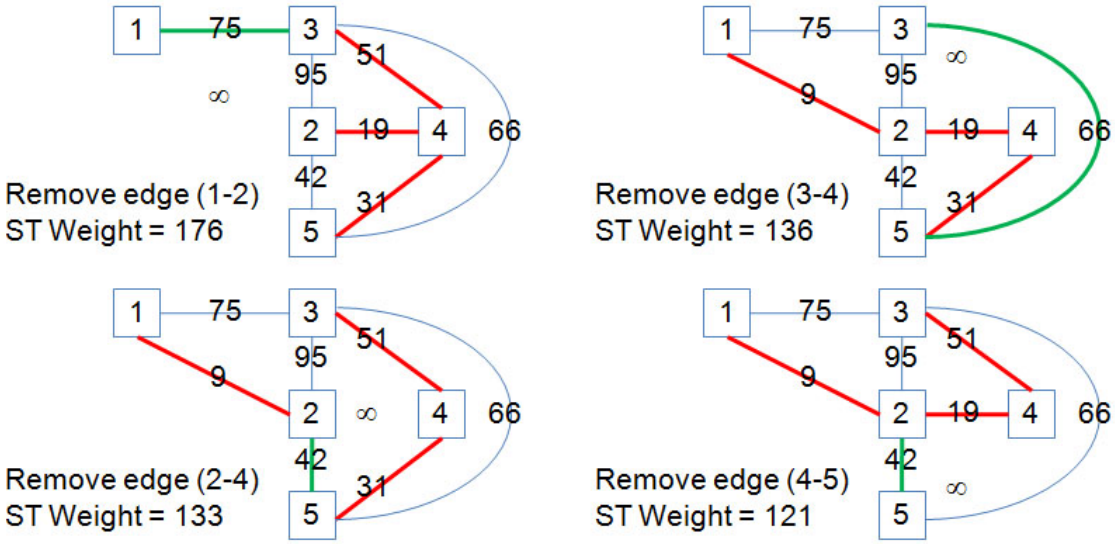


Figure 4.14: Finding the Second Best Spanning Tree from the MST

Programming Exercises for Kruskal's algorithm:

- Standard Application (for MST)
  1. UVa 908 - Re-connecting Computer Sites (discussed in this section)
  2. UVa 10034 - Freckles (straightforward MST problem)
  3. UVa 10307 - Killing Aliens in Borg Maze (build SSSP graph with BFS, then MST)
  4. UVa 11228 - Transportation System (split output for short versus long edges)
  5. UVa 11631 - Dark Roads (weight of all edges in graph - weight of all edges in MST)
  6. UVa 11710 - Expensive Subway (output 'Impossible' if graph still unconnected)
  7. UVa 11733 - Airports (maintain cost at every update)
  8. UVa 11747 - Heavy Cycle Edges (sum the edge weights of the chords)
  9. LA 4138 - Anti Brute Force Lock

- Variants
  1. UVa 10147 - Highways (Partial 'Minimum' Spanning Tree)
  2. UVa 10369 - Arctic Networks (Minimum Spanning 'Forest')
  3. UVa 10397 - Connect the Campus (Partial 'Minimum' Spanning Tree)
  4. UVa 10600 - ACM Contest and Blackout (Second Best Spanning Tree)
  5. UVa 10842 - Traffic Flow (find min weighted edge in 'Maximum' Spanning Tree)
  6. LA 3678 - The Bug Sensor Problem (Minimum Spanning 'Forest')
  7. LA 4110 - RACING ('Maximum' Spanning Tree)
  8. POJ 1679 - The Unique MST (Second Best Spanning Tree)

4.5 Dijkstra's

Motivating problem: Given a *weighted* graph G and a starting source vertex s, what are the *shortest paths* from s to the other vertices of G?

This problem is called the *Single-Source*[3] *Shortest Paths* (SSSP) problem on a *weighted graph*. It is a classical problem in graph theory and efficient algorithms exist. If the graph is unweighted, we can use the BFS algorithm as shown earlier in Section 4.3. For a general weighted graph, BFS does not work correctly and we should use algorithms like the $O((E+V)\log V)$ Dijkstra's algorithm (discussed here) or the $O(VE)$ Bellman Ford's algorithm (discussed in Section 4.6).

Edsger Wybe *Dijkstra*'s algorithm is a *greedy* algorithm: Initially, set the distance to all vertices to be `INF` (a large number) but set the `dist[source] = 0` (base case). Then, repeat the following process from the source vertex: From the current vertex u with the smallest `dist[u]`, 'relax' all neighbors of u. `relax(u, v)` sets `dist[v] = min(dist[v], dist[u] + weight(u, v))`. Vertex u is now done and will not be visited again. Then we greedily replace u with the unvisited vertex x with currently smallest `dist[x]`. See proof of correctness of this greedy strategy in [4].

There can be many ways to implement this algorithm, especially in using the `priority_queue`. The following code snippet may be one of the easiest implementation.

```
vector<int> dist(V, INF); dist[s] = 0; // INF = 2.10^9 not MAX_INT to avoid overflow
priority_queue<ii, vector<ii>, greater<ii> > pq; pq.push(ii(0, s)); // sort by distance

while (!pq.empty()) { // main loop
  ii top = pq.top(); pq.pop(); // greedy: pick shortest unvisited vertex
  int d = top.first, u = top.second;
  if (d == dist[u]) // This check is important! We want to process vertex u only once but we can
    // actually enqueue u several times in priority_queue... Fortunately, other occurrences of u
    // in priority_queue will have greater distances and can be ignored (the overhead is small) :)
    TRvii (AdjList[u], it) { // all outgoing edges from u
      int v = it->first, weight_u_v = it->second;
      if (dist[u] + weight_u_v < dist[v]) { // if can relax
        dist[v] = dist[u] + weight_u_v; // relax
        pq.push(ii(dist[v], v)); // enqueue this neighbor
}   } }                          // regardless whether it is already in pq or not
```

[3]This generic SSSP problem can also be used to solve: 1). Single-Pair Shortest Path problem where both source + destination vertices are given and 2). Single-Destination Shortest Paths problem where we can simply reverse the role of source and destination vertices.

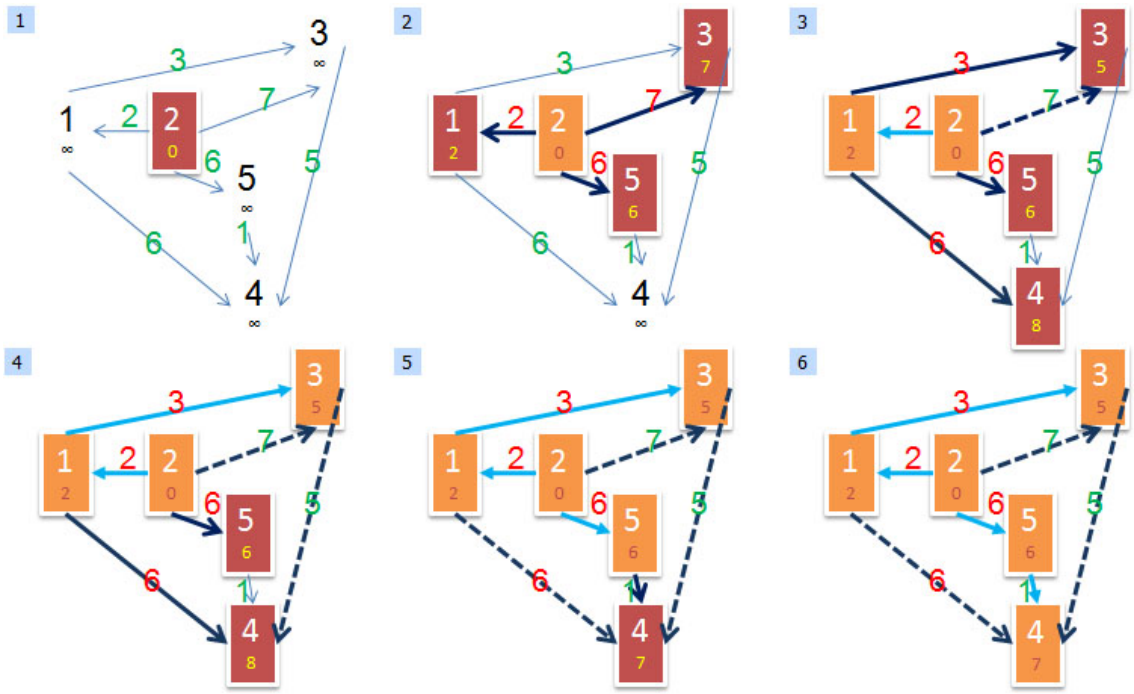


Figure 4.15: Dijkstra Animation on a Weighted Graph (from UVa 341 [17])

Figure 4.15 shows an example of running Dijkstra's on a simple weighted graph $|V| = 5$ and $|E| = 7$:

1. At the beginning, only `dist[source] = dist[2] = 0`, pq is {(0,2)}.
2. From vertex 2, we relax vertices {1, 3, 5}. Now `dist[1] = 2`, `dist[3] = 7`, and `dist[5] = 6`. Vertex 2 is done. The content of our `priority_queue` pq is {(2,1), (6,5), (7,3)}.
3. Among unprocessed vertices {1, 5, 3} in pq, vertex 1 has the least `dist[1] = 2` and is in front of pq. We dequeue (2,1) and relax all its neighbors: {3, 4} such that `dist[3] = min(dist[3], dist[1]+weight(1,3)) = min(7, 2+3) = 5` and `dist[4] = 8`. Vertex 1 is done. Now pq contains {(5,3), (6,5), (7,3), (8,4)}. See that we have 2 vertex 3. But it does not matter, as our Dijkstra's implementation will only pick one with minimal distance later.
4. We dequeue (5,3) and try to do `relax(3,4)`, but 5+5 = 10, whereas `dist[4] = 8` (from path 2-1-4). So `dist[4]` is unchanged. Vertex 3 is done and pq contains {(6,5), (7,3), (8,4)}.
5. We dequeue (6,5) and `relax(5, 4)`, making `dist[4] = 7` (the shorter path from 2 to 4 is now 2-5-4 instead of 2-1-4). Vertex 5 is done and pq contains {(7,3), (7,4), (8,4)}.
6. Now, (7,3) can be ignored as we know that `d > dist[3]` (i.e. $7 > 5$). Then (7,4) is processed as before. And finally (8,4) is ignored again as `d > dist[4]` (i.e. $8 > 7$). Dijkstra's stops here as the priority queue is empty.

4.6 Bellman Ford's

If the input graph has negative edge weight, Dijkstra can fail. Figure 4.16 is a simple example where Dijkstra's fails. Dijkstra's greedily sets `dist[3] = 3` first and uses that value to relax `dist[4] = dist[3]+weight(3,4) = 3+3 = 6`, before setting `dist[3] = dist[2]+weight(2,3) = 10+(-10) = 0`. The correct answer is `dist[4] = dist[3]+weight(3,4) = 0+3 = 3`.

To solve SSSP problem in the presence of negative edge weight, 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 4.8). This algorithm is simple: Relax all E edges $V-1$ times!

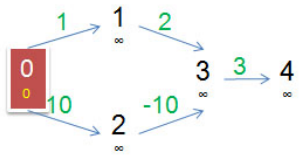


Figure 4.16: Dijkstra fails on Graph with negative weight

This is based on the idea that `dist[source] = 0` and if we relax an `edge(source, u)`, then `dist[u]` will have correct value. If we then relax an `edge(u, v)`, then `dist[v]` will also have 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 (path length: V-1 edges) should have been correctly computed. The code is simple:

```
vector<int> dist(V, INF); dist[s] = 0; // INF = 2B not MAX_INT to avoid overflow
REP (i, 0, V - 1) // relax all E edges V-1 times, O(V)
  REP (u, 0, V - 1) // these two loops = O(E)
    TRvii (AdjList[u], v) // has edge and can be relaxed
      dist[v->first] = min(dist[v->first], dist[u] + v->second);
REP (i, 0, V - 1)
  printf("SSSP(%d, %d) = %d\n", s, i, dist[i]);
```

The complexity of Bellman Ford's algorithm is $O(V^3)$ if the graph is stored as Adjacency Matrix or $O(VE)$ if the graph is stored as Adjacency List. This is (much) slower compared to Dijkstra's. Thus, Bellman Ford's is typically only used to solve SSSP problem when the input graph is not too big and *not guaranteed* to have all non-negative edge weights!

Bellman Ford's algorithm has one more interesting usage. After relaxing all E edges V-1 times, the SSSP problem should have been solved, i.e. there is no way we can relax any more vertex. This fact can be used to check the presence of *negative cycle*, although such a problem is ill-defined. In Figure 4.17, left, we see a simple graph with 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). But since there is a negative cycle, we can still do this one more time, i.e. relaxing `dist[1] = 946+15 = 961` $<$ `988`!

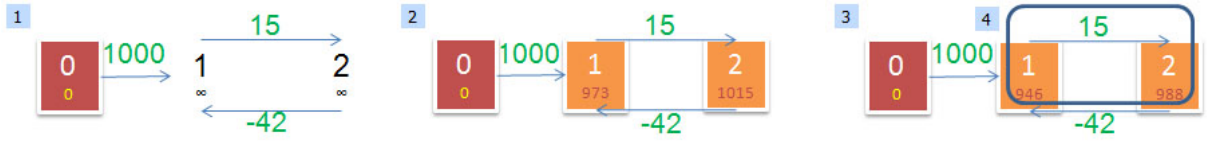


Figure 4.17: Bellman Ford's can detect the presence of negative cycle (from UVa 558 [17])

```
bool negative_cycle_exist = false;
REP (u, 0, V - 1) // one more pass to check
  TRvii (AdjList[u], v)
    if (dist[v->first] > dist[u] + v->second) // should be false, but if possible
      negative_cycle_exist = true;            // then negative cycle exists!
printf("Negative Cycle Exist? %s\n", negative_cycle_exist ? "Yes" : "No");
```

In Table 4.3, we present an SSSP algorithm decision table with programming contest in mind. This is to help readers in deciding which algorithm to choose depending on various graph criteria.

Graph Criteria	**BFS** $O(V+E)$	**Dijkstra's** $O((V+E)\log V)$	**Bellman Ford's** $O(VE)$
Unweighted	Best: $V, E \leq 1M$	Ok: $V, E \leq 50K$	Bad: $VE \leq 1M$
Weighted	WA except on Tree & DAG	Best: $V, E \leq 50K$	Ok: $VE \leq 1M$
Negative weight	WA	WA	Best: $VE \leq 1M$
Negative cycle	Cannot detect	Cannot detect	Can detect

Table 4.3: SSSP Algorithm Decision Table

Programming Exercises for Dijkstra's and Bellman Ford's algorithms:

- Dijkstra's Standard Application (for weighted SSSP)
  1. UVa 341 - Non-Stop Travel (actually solvable with Floyd Warshall's algorithm)
  2. UVa 929 - Number Maze (on a 2-D maze graph)
  3. UVa 10278 - Fire Station
  4. UVa 10603 - Fill
  5. UVa 10801 - Lift Hopping (model the graph carefully!)
  6. UVa 10986 - Sending email (straightforward Dijkstra's application)
  7. UVa 11377 - Airport Setup (model the graph carefully!)
  8. UVa 11492 - Babel (model the graph carefully!)
  9. UVa 11635 - Hotel Booking (Dijkstra's + BFS)
  10. LA 3290 - Invite Your Friends (+ BFS)
- Bellman Ford's Standard Application (for weighted SSSP with negative cycle)
  1. UVa 558 - Wormholes (checking the existence of negative cycle)
  2. UVa 10557 - XYZZY
  3. UVa 11280 - Flying to Fredericton (modified Bellman Ford's)

4.7 Floyd Warshall's

Basic Form and Application

Motivating Problem: Given a connected, weighted graph G with $V \leq 100$ 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. 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. Can we do better?

If the given graph is known to have $V \leq 100$, 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 3 nested loops. When it terminates, `AdjMatrix[i][j]` will contain the shortest path distance between two pair of vertices i and j in G. The original problem now become easy.

```
REP (k, 0, V - 1) // recall that #define REP(i, a, b) for (int i = int(a); i <= int(b); i++)
  REP (i, 0, V - 1)
    REP (j, 0, V - 1)
      AdjMatrix[i][j] = min(AdjMatrix[i][j], AdjMatrix[i][k] + AdjMatrix[k][j]);
```

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[4], but since $|V| \leq 100$ 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 $O((V+E)\ logV)$ 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 a programming contest setting, Floyd Warshall's main attractiveness is basically its implementation speed – 4 short lines only. If the given graph is small, do not hesitate using this algorithm – even if you only need a solution for the SSSP problem.

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.

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)` = weight of edge (i,j). We then find shortest paths between any two vertices with help of restricted intermediate vertices from vertex [0 ... k]. First, we only allow $k = 0$, then $k = 1$, ..., up to $k = V$-1.

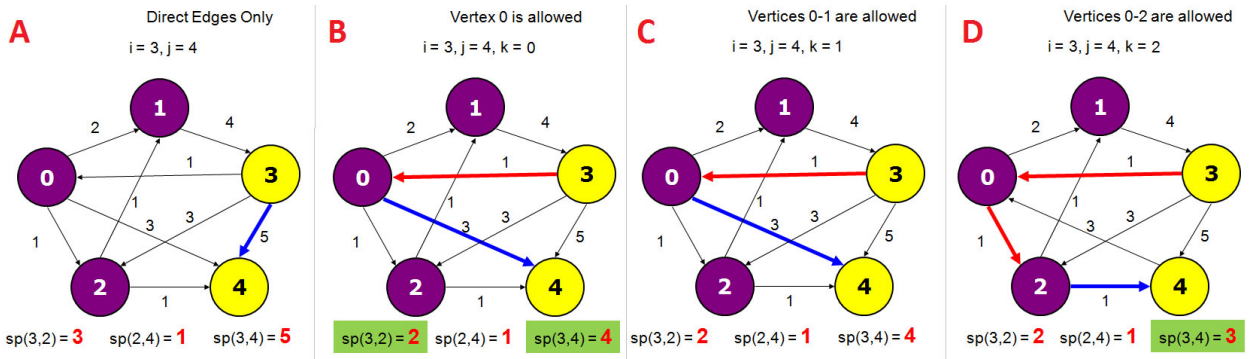


Figure 4.18: Floyd Warshall's Explanation

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

In Figure 4.18, we want to find `sp(3,4)`. The shortest possible path is 3-0-2-4 with cost 3. But how to reach this solution? We know that with direct edges only, `sp(3,4) = 5`, as in Figure 4.18.A. The solution for `sp(3,4)` will *eventually* be reached from `sp(3,2)+sp(2,4)`. But with only direct edges, `sp(3,2)+sp(2,4) = 3+1` is still > 3.

When we allow $k = 0$, i.e. vertex 0 can now be used as an intermediate vertex, then `sp(3,4)` is reduced as `sp(3,4) = sp(3,0)+sp(0,4) = 1+3 = 4`, as in Figure 4.18.B. Note that with $k = 0$, `sp(3,2)` – which we will need later – also drop from 3 to `sp(3,0)+sp(0,2) = 1+1 = 2`. Floyd Warshall's will process `sp(i,j)` 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), sp(3,2), nor to sp(2,4).

When we allow $k = 2$, i.e. vertices 0, 1, and 2 now can be used as the intermediate vertices, then `sp(3,4)` is reduced again as `sp(3,4) = sp(3,2)+sp(2,4) = 2+1 = 3`. Floyd Warshall's repeats this process for $k = 3$ and finally $k = 4$ but `sp(3,4)` remains at 3.

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 recurrence is as follows:
$D^{-1}_{i,j} = weight(i, j)$. This is the base case when we do not use any intermediate vertices.
$D^k_{i,j} = min(D^{k-1}_{i,j}, D^{k-1}_{i,k} + D^{k-1}_{k,j}) = min$(not using vertex k, using k), for $k \geq 0$, see Figure 4.19.

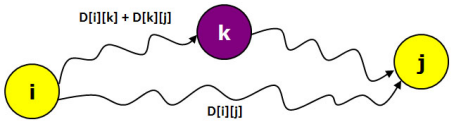


Figure 4.19: Using Intermediate Vertex to (Possibly) Shorten Path

This DP formulation requires us to fill the entries layer by layer. To fill out an entry in the table k, we make use of 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 Figure 4.20 for illustration.

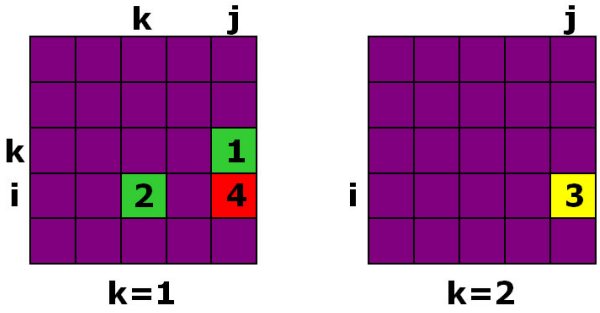


Figure 4.20: Floyd Warshall's DP Table

The naïve implementation is to use 3-dimensional matrix `D[k][i][j]` of size $O(V^3)$. However, we can utilize a space-saving trick by dropping dimension k and computing `D[i][j]` 'on-the-fly'. Thus, the Floyd Warshall's algorithm just need $O(V^2)$ space although it still runs in $O(V^3)$.

Other Applications

The basic purpose of Floyd Warshall's algorithm is to solve the APSP problem. However, it can also be applied to other graph problems.

Transitive Closure (Warshall's algorithm)

Stephen *Warshall* developed algorithm for finding solution for 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]`.

```
REP (k, 0, V - 1)
  REP (i, 0, V - 1)
    REP (j, 0, V - 1)
      d[i][j] = d[i][j] | (d[i][k] & d[k][j]);
```

Minimax and Maximin

The Minimax path problem is a problem of finding the minimum of maximum edge weight among all possible paths from i to j. There can be many paths from i to j. For a single path from i to j, we pick the maximum edge weight along this path. Then for all possible paths from i to j, pick the one with the minimum max-edge-weight. The reverse problem of Maximin is defined similarly.

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 min of either (itself) or (the max between `d[i][k]` or `d[k][j]`).

```
REP (k, 0, V - 1)
  REP (i, 0, V - 1)
    REP (j, 0, V - 1)
      d[i][j] = min(d[i][j], max(d[i][k], d[k][j]));
```

Exercise: In this section, we have shown you how to solve Minimax (and Maximin) with Floyd Warshall's algorithm. However, this problem can also be modeled as an MST problem and solved using Kruskal's algorithm. Find out the way!

Programming Exercises for Floyd Warshall's algorithm:

- Floyd Warshall's Standard Application (for APSP or SSSP on small graph)
  1. UVa 186 - Trip Routing (graph is small)
  2. UVa 341 - Non-Stop Travel (graph is small)
  3. UVa 423 - MPI Maelstrom (graph is small)
  4. UVa 821 - Page Hopping (one of the 'easiest' ICPC World Finals problem)
  5. UVa 10075 - Airlines (with special great-circle distances, see Section 7.2)
  6. UVa 10171 - Meeting Prof. Miguel (solution is easy with APSP information)
  7. UVa 11015 - 05-32 Rendezvous (graph is small)
  8. UVa 10246 - Asterix and Obelix
  9. UVa 10724 - Road Construction (adding one edge will only change 'few things')
  10. UVa 10793 - The Orc Attack (Floyd Warshall's simplifies this problem)
  11. UVa 10803 - Thunder Mountain (graph is small)
  12. UVa 11463 - Commandos (solution is easy with APSP information)

- Variants
  1. UVa 334 - Identifying Concurrent Events (transitive closure is only the sub-problem)
  2. UVa 534 - Frogger (Minimax)
  3. UVa 544 - Heavy Cargo (Maximin)
  4. UVa 869 - Airline Comparison (run Warshall's twice, then compare the AdjMatrices)
  5. UVa 925 - No more prerequisites, please!
  6. UVa 10048 - Audiophobia (Minimax)
  7. UVa 10099 - Tourist Guide (Maximin)

4.8 Edmonds Karp's

Basic Form and Application

Motivating problem: Imagine a connected, weighted, and directed graph as a pipe network where the pipes are the edges and the splitting points are the vertices. Each pipe has a capacity equals to the weight of the edge. 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 that can be passed by this pipe network)? This problem is called the Max Flow problem. An illustration of this problem is shown in Figure 4.21.

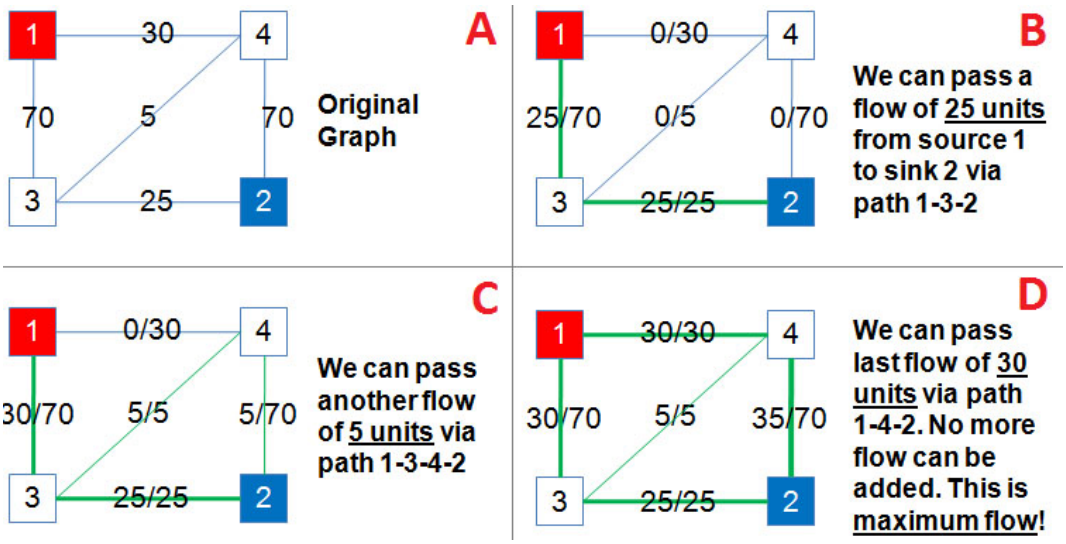


Figure 4.21: Illustration of Max Flow (From UVa 820 [17] - ICPC World Finals 2000 Problem E)

One solution is the Ford Fulkerson's method – invented by the same Lester Randolph *Ford*. Jr who created Bellman Ford's algorithm and Delbert Ray *Fulkerson*. The pseudo code is like this:

```
max_flow = 0
while (there exists an augmenting path p from s to t) { // iterative algorithm
  augment flow f along p, i.e.
    f = min edge weight in the path p
    max_flow += f // we can send flow f from s to t
    forward edges -= f // reduce capacity of these edges
    backward edges += f // increase capacity of reverse edges
}
output max_flow
```

There are several ways to find an augmenting path in the pseudo code above, each with different time complexity. In this section, we highlight two ways: via DFS or via BFS.

Ford Fulkerson's method implemented using DFS can run in $O(f^*E)$ where f^* is the max_flow value. This is because we can have a graph like in Figure 4.22 where every path augmentation only decreases the edge capacity along the path by 1. This is going to be repeated f^* times. In Figure 4.22, it is 200 times. The fact that number of edges in flow graph is $E \geq V - 1$ to ensure $\exists \geq 1$ s-t flow dictates that a DFS run is $O(E)$. The overall time complexity is $O(f^*E)$.

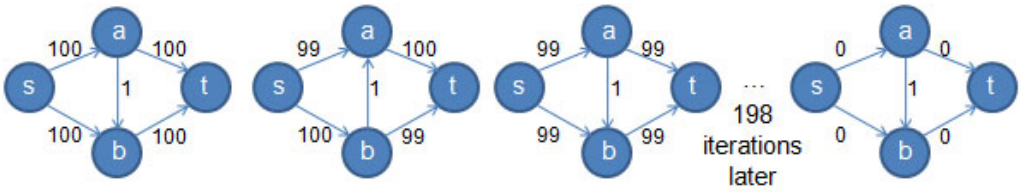


Figure 4.22: Implementation of Ford Fulkerson's Method with DFS is Slow

A better implementation of 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 *Karp*, thus named as Edmonds Karp's algorithm. It runs in $O(VE^2)$ as it can be proven that after $O(VE)$ iterations, all augmenting paths will already exhausted. Interested readers can browse books like [4] to study more about this algorithm. As BFS also runs in $O(E)$, the overall time complexity is $O(VE^2)$. Edmonds Karp's only needs two s-t paths in Figure 4.22: s-a-t (send 100 unit flow) and s-b-t (send another 100).

```
map<int, int> p; // parent map to reconstruct path
int f, s, t; // global variables

// inside int main()
int max_flow = 0;
while (1) { // this will be run max O(VE) times
  f = 0;

  queue<int> q; map<int, int> dist;  // O(E) BFS and record path p
  q.push(s); dist[s] = 0; // start from source
  while (!q.empty()) {
    int u = q.front(); q.pop(); // queue: layer by layer!
    if (u == t) break; // modification 1: reach sink t, stop BFS

    TRvii (AdjList[u], v) // for each neighbours of u
      // modification 2: also check AdjMat as edges may disappear
      if (AdjMat[u][v->first] > 0 && !dist.count(v->first)) {
        dist[v->first] = dist[u] + 1; // then v is reachable from u
        q.push(v->first); // enqueue v for next steps
        p[v->first] = u; // modification 3: parent of v->first is u
      }
  }

  augmentPath(t, INF); // path augmentation in O(V)
  if (f == 0) break; // seems that we cannot pass any more flow
  max_flow += f;
}
printf("Max flow = %d\n", max_flow);
```

```
void augmentPath(int v, int minEdge) {
  if (v == s) { // managed to get back to source
    f = minEdge; // minEdge of the path
    return;
  }
  else if (p.count(v)) { // augment if there is a path
    // we need AdjMat for fast lookup here
    augmentPath(p[v], min(minEdge, AdjMat[p[v]][v]));
    AdjMat[p[v]][v] -= f; // forward edges -> decrease
    AdjMat[v][p[v]] += f; // backward edges -> increase
} } // for more details why we must do that, consult references!
```

The code snippet above shows how to implement Edmonds Karp's algorithm in a way that it still achieves its $O(VE^2)$ time complexity. The code uses *both* Adjacency List (for fast enumeration of neighbors) and Adjacency Matrix (for fast access to edge weight) of the same flow graph.

In general, this $O(VE^2)$ Edmonds Karp's implementation is sufficient to answer most network flow problems in programming contests. However, for harder problems, we may need $O(V^2E)$ Dinic's or $O(V^3)$ Push-Relabel (relabel-to-front) Max Flow algorithms [4].

Exercise: The implementation of Edmonds Karp's algorithm shown here uses AdjMatrix to store residual capacity of each edge. A better way is to store flow of each edge, and then derive the residual from capacity of edge minus flow in edge. Modify the implementation!

Other Applications

There are several other interesting applications of Max Flow problem. We discuss six examples here while some others are deferred until Section 4.9. Note that some tricks shown in this section may be applicable to other graph problems.

Min Cut

Let's define an s-t cut $C = (S, T)$ as a partition of $V \in G$ such that source $s \in S$ and sink $t \in T$. Let's also define a *cut-set* of C to be the set $\{(u, v) \in E \mid u \in S, v \in T\}$ 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 Min Cut problem is to minimize the amount of capacity of an s-t cut. This problem is more general than finding bridges in a graph (See Section 4.2), i.e. in this case we can cut *more* than just one edge, but we want to do so in least cost way.

The solution for this problem is simple. The by-product of computing the Max Flow is Min Cut. In Figure 4.21.D, we can see that edges that are saturated, i.e. the flow on that edge equals to that edge's capacity, belong to the Min Cut!, i.e. edges 1-4 (capacity 30, flow 30), 3-4 (5/5) and 3-2 (25/25). The cost of cut is 30+5+25 = 60. This is the minimum over all possible s-t cuts. All vertices that are still reachable from source s belong to set S. The rest belong to set T. Here, $S = \{1, 3\}$ and $T = \{4, 2\}$.

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 tt with all t with infinite capacity, then run Edmonds Karp's algorithm as per normal.

Max Flow with Vertex Capacities

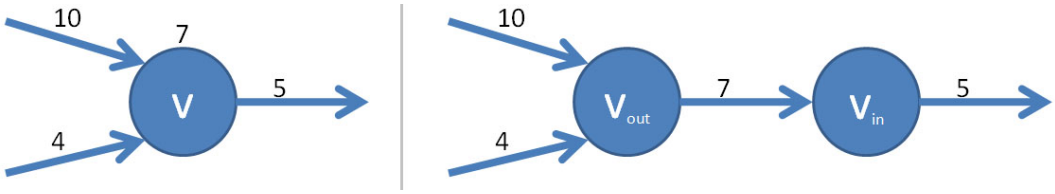


Figure 4.23: Vertex Splitting Technique

We can also have a Max Flow variant where 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 graph with a vertex weight can be converted into a more familiar graph without a vertex weight by splitting the vertex v to v_{out} and v_{in}, reassigning incoming edges to v_{out}/outgoing edges to v_{in}, and putting the original vertex v's weight as the weight of edge (v_{out}, v_{in}). For details, see Figure 4.23. Then run Edmonds Karp's algorithm as per normal.

Max 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 Edmonds Karp's algorithm as per normal.

Max Edge-Disjoint Paths

Finding the maximum number of edge-disjoint paths from s to t is similar to finding max 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 4.24 for a comparison between independent paths and edge-disjoint paths from $s = 1$ to $t = 7$.

Figure 4.24: Comparison Between 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. Min Cost (Max) Flow, or in short MCMF, can be solved by replacing the BFS to find augmenting path in Edmonds Karp's to Bellman Ford's (Dijkstra's may not work as the edge weight in the residual flow graph can be negative).

Programming Exercises for Ford Fulkerson's/Edmonds Karp's algorithm:

- Edmonds Karp's Standard Application (for Max Flow/Min Cut)
  1. UVa 820 - Internet Bandwidth (discussed in this section)
  2. UVa 10480 - Sabotage (Min Cut)
  3. UVa 10779 - Collector's Problem (Max Flow)
  4. UVa 11506 - Angry Programmer (Min Cut)
- Variants
  1. UVa 563 - Crimewave (Max Independent Paths: with unit edge and vertex capacities)
  2. UVa 10330 - Power Transmission (with Vertex Capacities, discussed in this section)
  3. UVa 10511 - Councilling
  4. UVa 10594 - Data Flow (basic MCMF problem)
  5. UVa 10806 - Dijkstra, Dijkstra. (Max Edge-Disjoint Paths + Min Cost)

4.9 Special Graphs

Some basic graph problems have simpler / faster polynomial algorithm if the given graph is *special*. So far we have identified the following special graphs that commonly appear in programming contests: **Tree**, **Directed Acyclic Graph (DAG)**, and **Bipartite Graph**. Problem setters may force contestants to use specialized algorithms for these special graphs simply by giving a large input size to judge a correct algorithm for general graph as Time Limit Exceeded (TLE). In this section, we will discuss some popular graph problems on these special graphs (see Figure 4.25).

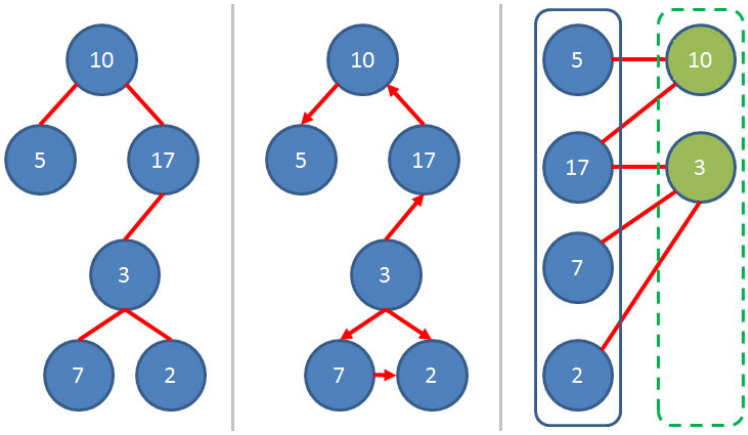


Figure 4.25: Special Graphs (Left to Right): Tree, Directed Acyclic Graph, Bipartite Graph

4.9.1 Tree

Tree is a special graph with the following characteristics: 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 from any pair of vertices.

Finding Articulation Points and Bridges in Tree

In Section 4.2, 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 4.5 and 4.6, we have seen two general purpose algorithms ($O((E + V) \log V)$ Dijkstra's and $O(VE)$ Bellman-Ford's) for solving the SSSP problem on a weighted graph. However, 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 path connecting the two vertices and the shortest path between these two vertices is basically the sum of edge weights of this unique path.

All-Pairs Shortest Paths on Weighted Tree

In Section 4.7, 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 process V times from each vertex, thus it is $O(V^2)$.

But this can still be improved to $O(V+Q\times L)$: Q is the number of query and L is the complexity of the Lowest Common Ancestor (LCA) implementation (see [40] for more details). We run $O(V)$ DFS/BFS once from any vertex v to find dist[v][other vertices] in tree. Then we can answer any shortest path query (i, j) on this tree by reporting dist[v][i] + dist[v][j] - 2 × dist[v][LCA(i, j)].

Diameter of Tree

The diameter of a graph is defined as the greatest distance between any pair of vertices. To find the diameter of a graph, we first find the shortest path between each pair of vertices (the APSP problem). The greatest length 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 4.7. However, if the given graph is a tree, the problem becomes simpler: do DFS/BFS from any node s to find furthest vertex x, 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. This solution only requires two calls of $O(V)$ graph traversal algorithm.

Max Weighted Independent Set on Tree

In Section 3.4, we have shown a DP on Tree example that solves Max Weighted Independent Set (MWIS) on Tree in $O(V)$. In Section 4.9.3 below, we will revisit this problem on a Bipartite Graph,

which can be reduced to Max Flow problem and runs in $O(VE^2)$ with Edmonds Karp's. However, this problem is NP-complete on general graph.

Programming Exercises related to Tree (also see Section 3.4.3 for 'DP on Tree' Topic):

1. UVa 112 - Tree Summing (backtracking)
2. UVa 115 - Climbing Trees (tree traversal, LCA)
3. UVa 122 - Trees on the level (tree traversal)
4. UVa 536 - Tree Recovery (tree traversal, reconstructing tree from pre + inorder)
5. UVa 615 - Is It A Tree? (graph property check)
6. UVa 699 - The Falling Leaves (preorder traversal)
7. UVa 712 - S-Trees (tree traversal)
8. UVa 10308 - Roads in the North (diameter of tree, discussed in this section)
9. UVa 10459 - The Tree Root (diameter + center of tree)
10. UVa 10701 - Pre, in and post (reconstructing tree from pre + inorder)
11. UVa 10938 - Flea Circus (use LCA)
12. UVa 11695 - Flight Planning (diameter + center of tree)

4.9.2 Directed Acyclic Graph

A Directed Acyclic Graph, abbreviated as DAG, is a special graph with the following characteristics: it is directed and has no cycle.

Single-Source Shortest Paths on DAG

The Single-Source Shortest Paths (SSSP) problem becomes much simpler if the given graph is a DAG as DAG has at least one topological order! We can use an $O(V+E)$ topological sort algorithm in Section 4.2 to find one such topological order, then relax 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. This way, the distance information propagation is correct with just one $O(V + E)$ linear pass!

Single-Source Longest Paths on DAG

Single-Source Longest Paths problem, i.e. finding the longest path from a starting vertex s is NP-complete on a general graph [39]. However the problem is again easy if the graph has no cycle, which is true in DAG. The solution for the Longest Paths in DAG (a.k.a. Critical Paths) is just a minor tweak from the SSSP solution in DAG shown above, i.e. simply negate all edge weights.

Min Path Cover on DAG

Motivating problem: Imagine that the vertices in Figure 4.26.A are passengers, and we draw an edge between two vertices $u-v$ if a single taxi can serve passenger u then passenger v on time. The question is: What is the minimum number of taxis that must be deployed to serve *all* passengers?

The answer for the motivating problem above is two taxis. In Figure 4.26.D, we see one possible solution. One taxi (red dotted line) serves passenger 1 (colored with red), passenger 2 (blue), and then passenger 4 (yellow). Another taxi (green dashed line) serves passenger 3 (green) and passenger 5 (orange). All passengers are served with just two taxis.

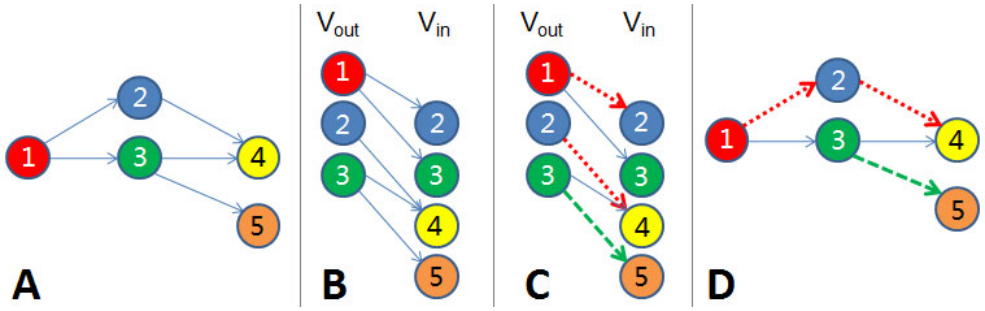


Figure 4.26: Min Path Cover in DAG (from LA 3126 [11])

In general, the Min Path Cover (MPC) problem in DAG is described as the problem of finding the minimum number of paths to cover *each vertex* in DAG $G = (V, E)$.

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 vertex itself). One matching between vertex a and vertex b using edge (a, b) says that we can use one less path as it can cover both vertices in $a \in V_{out}$ and $b \in V_{in}$. Thus if the Max Cardinality Bipartite Matching (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 [42].

Programming Exercises related to DAG:

- Single-Source Shortest/Longest Paths on DAG
  1. UVa 103 - Stacking Boxes
  2. UVa 10000 - Longest Paths
  3. UVa 10166 - Travel (shortest paths)
  4. UVa 10029 - Edit Step Ladders
  5. UVa 10350 - Liftless Eme (shortest paths)
  6. UVa 11324 - The Largest Clique (find SCC first then longest path on DAG)
  7. LA 3294 - The Ultimate Bamboo Eater (with 2-D Segment Tree)
  8. Ural/Timus OJ 1450 - Russian pipelines
  9. PKU 3160 - Father Christmas flymouse
- Counting Paths in DAG
  1. UVa 825 - Walking on the Safe Side (the graph is DAG, DP)
  2. UVa 926 - Walking Around Wisely (the graph is DAG, DP)
  3. UVa 988 - Many paths, one destination (topological sort + DP on DAG)

- Min Path Cover in DAG
  1. LA 2696 - Air Raid
  2. LA 3126 - Taxi Cab Scheme

4.9.3 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 $v \in V_2$. The most common application is the (bipartite) matching problem, shown below.

Max Cardinality Bipartite Matching

Motivating problem (from TopCoder [26] Open 2009 Qualifying 1): Group a list of numbers into pairs such that the sum of each pair is prime. For example, given the numbers {1, 4, 7, 10, 11, 12}, we can have: {1 + 4 = 5}, {1 + 10 = 11}, {1 + 12 = 13}, {4 + 7 = 11}, {7 + 10 = 17}, etc.

Actual task: Given a list of numbers N, return a list of all the elements in N that could be paired with $N[0]$ successfully as part of a *complete pairing* (i.e. each element a in N is paired to a unique other element b in N such that $a + b$ is prime), sorted in ascending order. The answer for the example above would be {4, 10} – omitting 12. This is because even though (1+12) is prime, there would be no way to pair the remaining 4 numbers whereas if we pair (1+4), we have (7+10), (11+12) and if we pair (1+10), we have (4+7), (11+12).

Constraints: list N contains an even number of elements (within $[2 \ldots 50]$, inclusive). Each element of N will be between 1 and 1000, inclusive. Each element of N will be distinct.

Although this problem involves finding prime numbers, this is not a pure 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 an option either because 50! is too big.

The key to solve this problem is to realize that this pairing or matching is done on *bipartite graph*! To get a prime number, we need to sum 1 odd + 1 even, because 1 odd + 1 odd = even number which is not prime, and 1 even + 1 even = also 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.

After we build this bipartite graph, the solution is trivial: If size of `set1` and `set2` are different, complete pairing is not possible. Otherwise, if the size of both sets is $n/2$, try to match `set1[0]` with `set2[k]` for $k = [0 \ldots n/2 - 1]$ and do Max Cardinality Bipartite Matching (MCBM) for the rest. This problem can be solved with Max Flow algorithm like $O(VE^2)$ Edmonds Karps algorithm. If we obtain $n/2 - 1$ more matchings, add `set2[k]` to the answer. For this test case, the answer is {4, 10}.

Bipartite Matching 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 4.27).

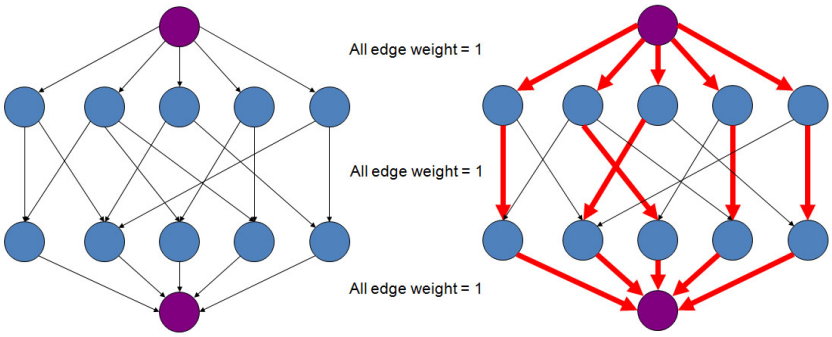


Figure 4.27: Bipartite Matching can be reduced to Max Flow problem

Max Weighted Independent Set on Bipartite Graph

Motivating Problem: Suppose that there are two users: User A and B. Each user has transactions, e.g. A has $\{A_1, A_2, \ldots, A_n\}$ and each transaction has a weight, e.g. $W(A_1), W(A_2)$, etc. These transactions use shared resources, e.g. transaction $A1$ uses $\{r_1, r_2\}$. Access to a resource is exclusive, e.g. if A_1 is selected, then any of user B's transaction(s) that use either r_1 or r_2 cannot be selected. It is guaranteed that two requests from user A will *never* use the same resource, but two requests from different users may be competing for the same resource. Our task is to maximize the sum of weight of the selected transactions!

Let's do several keyword analysis of this problem. If a transaction from user A is selected, then transactions from user B that share some or all resources cannot be selected. This is a strong hint for **Independent Set**. And since we want to maximize sum of weight of selected transactions, this is **Max Weighted Independent Set (MWIS)**. And since there are only two users (two sets) and the problem statement guarantees that there is no resource conflict between the transactions from within one user, we are sure that the input graph is a **Bipartite Graph**. Thus, this problem is actually an **MWIS on Bipartite Graph**.

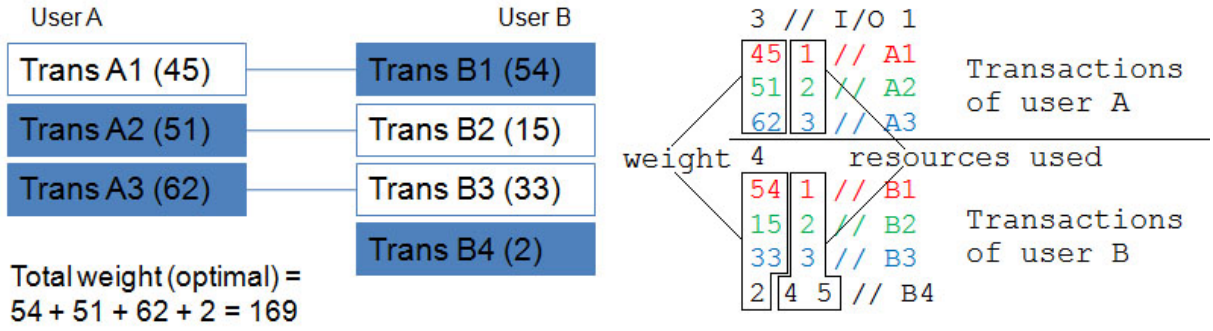
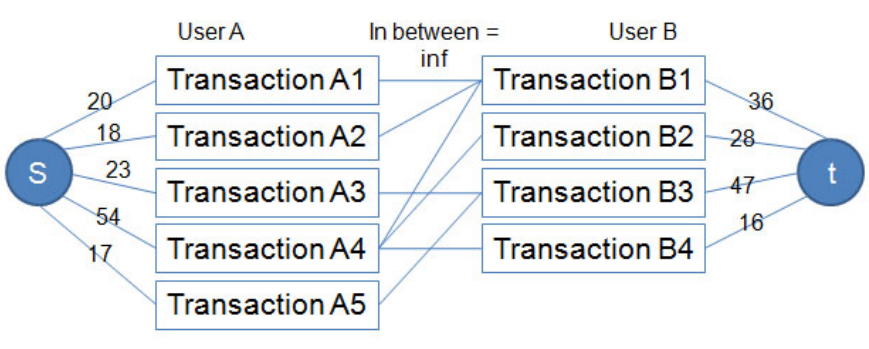


Figure 4.28: Example of MWIS on Bipartite Graph (from LA 3487 [11])

Let's see Figure 4.28 for illustration. We have two users. We list down all transactions of A on the left and all transactions of B on the right. We draw an edge between two transactions if they share similar resource. For example, transaction $A1$ uses resource 1 and transaction $B1$ also uses resource 1. We draw an edge between $A1$ (with weight 45) and $B1$ (with weight 54) because they share the same resource. In fact, there are two more edges between $A2 - B2$ and $A3 - B3$. Transaction $B4$ has no edge because the resources that it used: {4, 5} are not shared with any other transactions. In this instance, $\{B_1(54), A_2(51), A_3(62), B_4(2)\}$ is the MWIS, with total weight = 54+51+62+2 = 169.

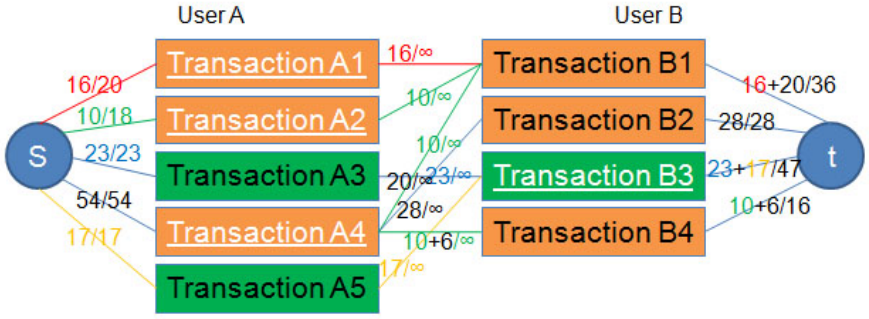
To find the solution for non-trivial cases, we have to reduce this problem to a Max Flow problem. We assign the original vertex cost (the weight of taking that vertex) as capacity from source to that vertex for user A and capacity from that vertex to sink for user B. Then, we give 'infinite' capacity in between any edge in between sets A and B. See Figure 4.29.




```
5 // I/O 2
20 1 // A1
18 2 // A2
23 4 // A3
54 3 5 6 // A4
17 7 // A5
4
36 1 2 3 // B1
28 5 // B2
47 4 7 // B3
16 6 // B4
```

Figure 4.29: Reducing MWIS on Bipartite Graph to Max Flow Problem (from LA 3487 [11])

Then, we run $O(VE^2)$ Edmonds Karp's algorithm on this Flow graph. After the Max Flow algorithm terminates, the solution is {s-component $\bigcap$ vertices in User A} + {t-component $\bigcap$ vertices in User B} where s-component (t-component) are the vertices still reachable to source vertex (sink vertex) after running Max Flow. In Figure 4.30, the solution is: $\{A_1(20), A_2(18), A_4(54)\}+\{B_3(47)\} = 139$. This value can also be obtained via: MWIS = Total Weight - Max Flow = $259 - 120 = 139$.




```
5 // I/O 2
20 1 // A1
18 2 // A2
23 4 // A3
54 3 5 6 // A4
17 7 // A5
4
36 1 2 3 // B1
28 5 // B2
47 4 7 // B3
16 6 // B4
```

Figure 4.30: Solution for Figure 4.29 (from LA 3487 [11])

Programming Exercises related to Bipartite Graph:

- Maximum Cardinality Bipartite Matching (graph modeling + Max Flow)
  1. UVa 670 - The Dog Task
  2. UVa 753 - A Plug for Unix
  3. UVa 10080 - Gopher II
  4. UVa 10092 - The Problem with the Problemsetter
  5. UVa 10735 - Euler Circuit
  6. UVa 11045 - My T-Shirt Suits Me
  7. UVa 11418 - Clever Naming Patterns
  8. Top Coder Open 2003 Semifinal Round 4 - Division 1, Level 3 - RookAttack
  9. Top Coder Open 2009: Qualifying 1 - Prime Pairs
  10. LA 4407 - Gun Fight

- Max Weighted Independent Set in Bipartite Graph
  1. UVa 11159 - Factors and Multiples (similar solution with Bipartite Matching above)
  2. LA 3487 - Duopoly (also in Zhejiang Online Judge problem 2129)
- Max Vertex Cover in Bipartite Graph
  1. UVa 11419 - SAM I AM (Min Vertex Cover)
  2. LA 2523 - Machine Schedule (also in PKU 1325 - Machine Schedule)
  3. PKU 2226 - Muddy Fields

4.10 Chapter Notes

Take note 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, to combine longest path in DAG with Segment Tree data structure, etc.

This chapter, albeit already quite long, still omits many known graph algorithms and graph problems that are sometimes tested in ICPCs or IOIs, namely: **Kosaraju's** algorithm for finding Strongly Connected Component, **Prim's** and **Boruvka's** algorithms for Minimum Spanning Tree, k-th shortest paths, **Euler's** Path/Tour, **Fleury's** algorithm, **Chinese Postman Problem**, **Hamiltonian** Path/Tour, Bitonic Traveling Salesman Problem, Arborescence, **Tarjan's Offline Lowest Common Ancestor**, **Dinic's** or **Push Relabel** algorithms for Max Flow, Circulation Problem, **Kuhn Munkres's (Hungarian)** matching algorithm, **Edmonds's Blossom Shrinking**, etc.

If you want to increase your winning chance in ACM ICPC, please spend some time to study them beyond this book. These harder ones rarely appears in *regional* contests and if they are, they usually become the *decider* problem. Harder graph problems like these are more likely to appear in ACM ICPC World Finals level.

However, for IOI contestants, most graph materials in IOI syllabus are already covered in this chapter.

There are approximately **173 programming exercises** discussed in this chapter.

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

Recent ICPCs (especially in Asia) usually contain one or two mathematics problems. This chapter aims to prepare contestants in dealing with them.

5.1 Overview and Motivation

As with the topic of graph in previous chapter, there exist mathematics problems in recent ICPC problem sets – at least one, and can be two per problem set (see Table 5.1)!

LA	Problem Name	Source	Mathematics Problem (Algorithm)
2194	The Luncheon	Dhaka06	
2195	Counting Zeroes	Dhaka06	Prime Numbers
2953	Sum of Factorials	Guangzhou03	
2955	Vivian's Problem	Guangzhou03	
3172	Period of an Infinite ...	Manila06	
3399	Sum of Consecutive ...	Japan05	Prime Numbers
3904	Tile Code	Seoul07	Combinatorics
3997	Numerical surprises	Danang07	
4104	MODEX	Singapore07	Modular Exponentiation (Java BigInteger)
4203	Puzzles of Triangles	Dhaka08	
4209	Stopping Doom's Day	Dhaka08	Formula Simplification + BigInt
4270	Discrete Square Roots	Hefei08	
4340	Find Terrorists	Amrita08	
4406	Irreducible Fractions	KLumpur08	
4715	Rating Hazard	Phuket09	Farey Sequence
4721	Nowhere Money	Phuket09	Fibonacci, Zeckendorf Theorem

Table 5.1: Some Mathematics Problems in Recent ACM ICPC Asia Regional

The appearance of mathematics 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.

We are aware that different countries have different emphasis in mathematics training in pre-University education. Thus, for some newbie ICPC contestants, the term 'Euler Phi' is a familiar term, but for others, the term does 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 common mathematic terminologies, definitions, problems, and algorithms that frequently appear in programming contests.

5.2 Ad Hoc Mathematics Problems

We start this chapter by mentioning Ad Hoc mathematics problems. They are basically contest problems involving mathematics that requires no more than basic programming skills.

Programming Exercises related to Ad Hoc Mathematics:

1. UVa 344 - Roman Numerals (conversion from roman numerals to decimal and vice versa)
2. UVa 377 - Cowculations (base 4 operations)
3. UVa 10346 - Peter's Smoke (simple math)
4. UVa 10940 - Throwing Cards Away II (find pattern using brute force, then use the pattern)
5. UVa 11130 - Billiard bounces (use billiard table reflection technique)
6. UVa 11231 - Black and White Painting (use the $O(1)$ formula once you spot the pattern)
7. UVa 11313 - Gourmet Games (similar to UVa 10346)
8. UVa 11428 - Cubes (simple math with complete search)
9. UVa 11547 - Automatic Answer (one liner $O(1)$ solution exists)
10. UVa 11723 - Numbering Road (simple math)
11. UVa 11805 - Bafana Bafana (very simple $O(1)$ formula exists)

5.3 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.

5.3.1 Prime Numbers

A natural number starting from 2: $\{2, 3, 4, \ldots\}$ 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 [20]). There are 25 primes in range $[0 \ldots 100]$, 168 primes in $[0 \ldots 1000]$, 1000 primes in $[0 \ldots 7919]$, 1229 primes in $[0 \ldots 10000]$, etc...

Prime number is an important topic in number theory and the source for many programming problems[1]. 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* $\in [2 \dots N\text{-}1]$. 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* $\in [2 \dots \sqrt{N}]$, i.e. we stop when the *divisor* is already greater than $\sqrt{N}$. This is because 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 previous version, but can still be improved to be twice faster.

The second improvement is to test if N is divisible by *divisor* $\in [3, 5, 7 \dots \sqrt{N}]$, 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[2] 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 $[1 \dots \sqrt{(10^6)}]$, but there are only 168 primes in the same range. The number of primes $\leq M$ – denoted by $\pi(M)$ – is bounded by $O(M/(\ln(M) - 1))$, so 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 $[0 \dots N]$, there is a better algorithm than testing each number in the range whether it is a prime 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[3] of 2 starting from $2 \times 2 = 4$, 6, 8, 10, ... until it 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, 18, ... Then it takes 5 and crosses out all multiples of 5 starting from $5 \times 5 = 25$, 30, 35, 40, ... After that, whatever left uncrossed within the range $[0 \dots N]$ 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* $O(N \log \log N)$ [44].

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 sieve for smaller primes and reserve optimized prime testing function for larger primes – see previous discussion. The combined code is as follows:

[1]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**.

[2]This is a bit recursive – testing whether a number is a prime by using another (smaller) prime numbers. But the reason should be obvious after reading the next section.

[3]Common sub-optimal implementation is to start from 2 × i instead of i × i, but the difference is not that much.

```
#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 + small extra bits should be enough for most prime-related problems
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.reset(); bs.flip(); // set all numbers to 1
  bs.set(0, false); bs.set(1, false); // except index 0 and 1
  for (ll i = 2; i <= _sieve_size; i++) if (bs.test((size_t)i)) {
    // cross out multiples of i starting from i * i!
    for (ll j = i * i; j <= _sieve_size; j += i) bs.set((size_t)j, false);
    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.test(N); // O(1) for small primes
  REP (i, 0, primes.size() - 1) 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

// in int main()
sieve(10000000); // can go up to 10^7
printf("%d\n", isPrime(5915587277)); // 10 digit prime
```

Optimized Trial Divisions for Finding Prime Factors

In number theory, we know that a prime number N only have 1 and itself as factors but a **composite** numbers N, i.e. the non-primes, can be written uniquely it as a multiplication of its prime factors. That's it, prime numbers are multiplicative building blocks of integers. 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 how many of them 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 $N' = 1$. Special case if N is actually a prime number. The code template below takes in an integer N and return the list of prime factors – see code below.

```
vi primeFactors(int N) {
  vi factors;          // vi "primes" (generated by sieve) is optional
  int PF_idx = 0, PF = primes[PF_idx]; // using PF = 2, 3, 4, ..., 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;
}
```

```
// in int main()
sieve(100); // prepare list of primes [0 .. 100]
vi result = primeFactors(10000); // with that, we can factor up to 100^2 = 10000
vi::iterator last = unique(result.begin(), result.end()); // to remove duplicates
for (vi::iterator i = result.begin(); i != last; i++) // output: 2 and 5
  printf("%d\n", *i);
```

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 $O(\pi(\sqrt{N})) = O(\sqrt{N}/ln\sqrt{N})$. However, if given composite numbers, this algorithm is reasonably fast.

Programming Exercises about Prime Numbers problems:

1. UVa 294 - Divisors
2. UVa 406 - Prime Cuts
3. UVa 516 - Prime Land
4. UVa 524 - Prime Ring Problem (requires backtracking)
5. UVa 543 - Goldbach's Conjecture
6. UVa 583 - Prime Factors
7. UVa 686 - Goldbach's Conjecture (II)
8. UVa 897 - Annagramatic Primes
9. UVa 914 - Jumping Champion
10. UVa 993 - Product of digits
11. UVa 10006 - Carmichael Numbers
12. UVa 10042 - Smith numbers
13. UVa 10140 - Prime Distances
14. UVa 10200 - Prime Time
15. UVa 10235 - Simply Emirp (reverse prime)
16. UVa 10311 - Goldbach and Euler
17. UVa 10394 - Twin Primes (adjacent primes)
18. UVa 10533 - Digit Primes
19. UVa 10539 - Almost Prime Numbers
20. UVa 10637 - Coprimes
21. UVa 10650 - Determinate Prime (find 3 consecutive primes that are uni-distance)
22. UVa 10699 - Count the Factors
23. UVa 10738 - Riemann vs. Mertens
24. UVa 10789 - Prime Frequency
25. UVa 10852 - Less Prime
26. UVa 10924 - Prime Words
27. UVa 10948 - The Primary Problem
28. UVa 11287 - Pseudoprime Numbers
29. UVa 11408 - Count DePrimes
30. UVa 11466 - Largest Prime Divisor

5.3.2 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 fraction, 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 [20, 4] 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 [20] 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 before 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 332 - Rational Numbers from Repeating Fractions
2. UVa 412 - Pi
3. UVa 530 - Binomial Showdown
4. UVa 10193 - All You Need Is Love
5. UVa 10407 - Simple Division
6. UVa 10680 - LCM
7. UVa 10717 - Mint (requires Complete Search on top of LCM/GCD)
8. UVa 10791 - Minimum Sum LCM (prime factorization + tricky cases)
9. UVa 10892 - LCM Cardinality
10. UVa 11388 - GCD LCM (must understand the relationship between GCD and LCM)
11. UVa 11417 - GCD

5.3.3 Euler's Totient (Phi) Function

Now that we have discussed prime number and GCD, we can define **relatively prime**. Two integers a and b are said to be relatively prime if $gcd(a, b) = 1$, e.g. 25 and 42.

There is an interesting problem of finding positive integers below N that are relatively prime to N (see programming exercises below). A naïve algorithm starts with `counter = 0`, iterates through $i \in [1 \ldots N]$, and increases the `counter` if $gcd(i, N) = 1$, but this is slow for a large N.

A better algorithm is the Euler's Totient (Phi) function [36]. 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 we iterate through all prime factors of N. For example $N = 36 = 2^2 \times 3^2$. $\varphi(36) = 36 \times (1 - \frac{1}{2}) \times (1 - \frac{1}{3}) = 12$.

In fact, there are only 12 integers less than equal to 36 that are relatively prime to 36. They are 1, 5, 7, 11, 13, 17, 19, 23, 25, 29, 31, and 35. As we need to factor N, the complexity of this algorithm is similar with the complexity of factoring an integer with trial division mentioned earlier. The code is below.

```
int EulerPhi(int N) {
  vi factors = primeFactors(N);
  vi::iterator new_end = unique(factors.begin(), factors.end()); // get unique
  int result = N;
  for (vi::iterator i = factors.begin(); i != new_end; i++)
    result = result - result / *i;
  return result;
}
```

Programming Exercises related to Euler's Totient (Phi) Function:

1. UVa 10179 - Irreducible Basic Fractions (direct application of Euler's Phi function)
2. UVa 10299 - Relatives (another direct application of Euler's Phi function)
3. UVa 10820 - Send A Table (a[i] = a[i - 1] + 2 $\times\varphi$(i))
4. UVa 11064 - Number Theory
5. UVa 11327 - Enumerating Rational Numbers

5.3.4 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 fruit does she buys?

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! This is different from System of Linear Equations discussed later in Section 5.5.5. The solution for the Linear Diophantine Equation is as follow [20].

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 (also see [4], Chapter 31), and the rest can be derived from $x = x_0 + (b/d)n$, $y = y_0 - (a/d)n$, where n is an integer.

```
// 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);
  int x1 = y;
  int y1 = x - (a / b) * y;
  x = x1;
  y = y1;
}
```

Using `extendedEuclid`, the Linear Diophantine Equation with two variables can be easily solved. For our motivating problem above: $25x + 18y = 839$, we have:
$a = 25, b = 18, extendedEuclid(25, 18) = ((-5, 7), 1)$, 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,
$x = -4195 + (18/1)n, y = 5873 - (25/1)n$.
Since we need to have non-negative x and y, we have:
$-4195 + 18n \geq 0$ 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 234.
Thus $x = -4195 + 18 \times 234 = 17$ 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 algorithm:

1. UVa 718 - Skycraper Floors
2. UVa 10090 - Marbles (use solution for Linear Diophantine Equation)
3. UVa 10104 - Euclid Problem (pure problem involving Extended Euclid)

5.3.5 Modulo Arithmetic

Some mathematics computations in programming problems can end up having very large positive (or very small negative) results that reside the range of largest integer data type (currently the 64-bit `long long` in C++). Sometimes, we are only interested with the result modulo a number.

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. 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), is divisible by 131071.

Programming Exercises related to Modulo Arithmetic:

1. UVa 374 - Big Mod
2. UVa 602 - What Day Is It?
3. UVa 10174 - Couple-Bachelor-Spinster Numbers
4. UVa 10176 - Ocean Deep! Make it shallow!! (as discussed above)
5. UVa 10212 - The Last Non-zero Digit
6. UVa 10489 - Boxes of Chocolates
7. LA 4104 - MODEX

5.3.6 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, ... which can be derived with an $O(n)$ DP technique. 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, etc). Note that Fibonacci sequence grows very fast and sometime problems involving Fibonacci have to be solved using Java BigInteger library (see Section 5.4 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.

Programming Exercises related to Fibonacci:

1. UVa 495 - Fibonacci Freeze (use Java BigInteger class)
2. UVa 763 - Fibinary Numbers (Zeckendorf representation)
3. UVa 900 - Brick Wall Patterns (Combinatorics, the pattern is similar to Fibonacci)
4. UVa 948 - Fibonaccimal Base (Zeckendorf representation)
5. UVa 10183 - How many Fibs?
6. UVa 10229 - Modular Fibonacci
7. UVa 10334 - Ray Through Glasses (use Java BigInteger class)
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
11. UVa 10862 - Connect the Cable Wires (the pattern ends up very similar to Fibonacci)
12. UVa 11000 - Bee
13. UVa 11161 - Help My Brother (II) (Fibonacci + median)
14. UVa 11780 - Miles 2 Km (the background of this problem is about Fibonacci Numbers)

5.3.7 Factorial

Factorial of n, i.e. $fac(n)$ is defined as 1 if $n = 0$ or $n \times fac(n-1)$ if $n > 0$. Factorial also grows very fast and may also need Java BigInteger library (Section 5.4).

Programming Exercises related to Factorial:

1. UVa 160 - Factors and Factorials
2. UVa 324 - Factorial Frequencies
3. UVa 568 - Just the Facts
4. UVa 623 - 500! (use Java BigInteger class)
5. UVa 884 - Factorial Factors

6. UVa 10061 - How many zeros & how many digits? (there exists a formula for this)
7. UVa 10139 - Factovisors (there exists a formula for this)
8. UVa 10858 - Recover Factorial
9. UVa 10220 - I Love Big Numbers! (use Java BigInteger class)
10. UVa 10323 - Factorial! You Must Be Kidding
11. UVa 10780 - Again Prime? No time.
12. UVa 11347 - Multifactorials
13. UVa 11415 - Count the Factorials

5.4 Java BigInteger Class

5.4.1 Basic Features

When intermediate and/or final results of an integer-based mathematics computation cannot be stored in the largest built-in integer data type and the given problem does not use any modulo arithmetic technique, we have to resort to BigInteger (sometimes also called as BigNum) libraries. A simple example: try to compute 25! (factorial of 25). The result is 15,511,210,043,330,985,984,000,000 which is beyond the capacity of 64-bit `unsigned long long` in C/C++.

The way the BigInteger library works is to store the big integer as a (long) string. For example we can store 10^{21} inside a string `num1` = "1,000,000,000,000,000,000,000" without problem whereas this is already overflow in a 64-bit `unsigned long long` in C/C++. Then, for common mathematical operations, the BigInteger library uses a kind of digit by digit operations to process the two big integer operands. For example with `num2` = "17", we have `num1 * num2` as:

```
num1         =  1,000,000,000,000,000,000,000
num2         =                             17
               ------------------------------ *
                7,000,000,000,000,000,000,000
               10,000,000,000,000,000,000,00
               ------------------------------ +
num1 * num2 = 17,000,000,000,000,000,000,000
```

Addition and subtraction are two simpler operations in BigInteger. Multiplication takes a bit more programming job. Efficient division and raising number 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. Fortunately, Java has a BigInteger class that we can use for this purpose (as of 9 August 2010, C++ STL currently does not have such library thus it is a good idea to use Java for BigInteger problems).

The Java BigInteger (BI) class supports the following basic integer operations: addition – `add(BI)`, subtraction – `subtract(BI)`, multiplication – `multiply(BI)`, division – `divide(BI)`, remainder – `remainder(BI)`, combination of division and remainder – `divideAndRemainder(BI)`, modulo – `mod(BI)` (slightly different to `remainder(BI)`), and power – `pow(int exponent)`. For example, the following short Java code is the solution for UVa 10925 - Krakovia which simply requires BigInteger addition (to sum N large bills) and division (to divide the large sum to F friends).

```
import java.io.*;
import java.util.*; // Scanner class is inside this package
import java.math.*; // BigInteger class is inside this package

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.valueOf(0); // use valueOf to initialize
      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
```

5.4.2 Bonus Features

The 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 `toString(int radix)`.

GCD Revisited

When we need to compute the GCD of two **big** integers, we do not have to worry. See an example below for UVa 10814 - Simplifying Fractions that ask us to simplify a given (large) fraction to its simplest form by dividing both numerator and denominator with the gcd between them.

```
import java.io.*;
import java.util.*;
import java.math.*;

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) {
      BigInteger p = sc.nextBigInteger();
      String ch = sc.next(); // ignore this one
      BigInteger q = sc.nextBigInteger();
      BigInteger gcd_pq = p.gcd(q); // wow :)
      System.out.println(p.divide(gcd_pq) + " / " + q.divide(gcd_pq));
} } }
```

Modulo Arithmetic Revisited

One of the problems presented in the previous section is LA 4104 - MODEX. We are asked to find what is the value of $x^y \pmod{n}$. It turns out that this problem can be solved with:

```
import java.io.*;
import java.util.*;
import java.math.*;

class Main { /* LA 4104 - MODEX */
  public static void main(String[] args) {
    Scanner sc new Scanner(System.in);
    int nTC = sc.nextInt();
    while (nTC-- > 0) {
      BigInteger x = BigInteger.valueOf(scan.nextInt());
      BigInteger y = BigInteger.valueOf(scan.nextInt());
      BigInteger n = BigInteger.valueOf(scan.nextInt());
      System.out.println(x.modPow(y, n)); // look ma, it's in the library ;)
} } }
```

Base Number Conversion

The base number conversion is actually a not-so-difficult[4] mathematical problem, but this problem becomes even simpler with Java BigInteger class. We can construct and print a big integer in any base (radix) as shown below:

```
import java.io.*;
import java.util.*;
import java.math.*;

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 constructor!
      String m_str = sc.next();
      BigInteger m = new BigInteger(m_str, b); // 2nd parameter is the radix/base
      System.out.println((p.mod(m)).toString(b)); // can output in any radix/base
} } }
```

Programming Exercises related to Big Integer that are **not** mentioned elsewhere in this chapter.

1. UVa 343 - What Base Is This? (number base conversion)
2. UVa 355 - The Bases Are Loaded (number base conversion)
3. UVa 389 - Basically Speaking (number base conversion)
4. UVa 424 - Integer Inquiry (bignum addition)
5. UVa 446 - Kibbles 'n' Bits 'n' Bits 'n' Bits (number base conversion)
6. UVa 636 - Squares (number base conversion++)
7. UVa 10083 - Division (bignum + number theory)
8. UVa 10551 - Basic Remains (bignum mod and base conversion)

[4]For example: $(132)_8$ is $1 \times 8^2 + 3 \times 8^1 + 2 \times 8^0 = 64 + 24 + 2 = (90)_{10}$ and $(90)_{10}$ is 90 (keep dividing by 2) $\rightarrow 45(0) \rightarrow 22(1) \rightarrow 11(0) \rightarrow 5(1) \rightarrow 2(1) \rightarrow 1(0) \rightarrow 0(1) = (1011010)_2$ (read backwards).

9. UVa 10814 - Simplifying Fractions (bignum gcd)
10. UVa 10106 - Product (bignum multiplication)
11. UVa 10473 - Simple Base Conversion (number base conversion)
12. UVa 10523 - Very Easy!!! (bignum addition, multiplication, and power)
13. UVa 10925 - Krakovia (bignum addition and division)
14. LA 4104 - MODEX (bignum modPow)

5.5 Miscellaneous Mathematics Problems

In this section, we list down a few more mathematics problems that have been used a few times in some programming contests. Mastering the solutions for these problems can be an advantage if you are given similar problem or problem that uses these knowledge as part of solution for the problem. The terms mentioned here also act as *keywords* for further study.

5.5.1 Combinatorics

Combinatorics [31] is a branch of *discrete* mathematics concerning the study of finite or **countable** discrete structures. Programming problems involving combinatorics usually titled 'How Many [Object]', 'Count [Object]', etc, although some problem setters choose to hide this fact from their problem title. The code is usually short, but finding the recurrence formula takes some mathematics brilliance and patience. In ICPC, if such problem exists in the given problem set, ask one team member to derive the formula whereas the other two concentrates on other problems. Quickly code the usually short formula once it is obtained.

For example, try solving UVa 10401 - Triangle Counting. This problem has a short description: "given n rods of length 1, 2, ..., n, pick any 3 of them & build a triangle. How many distinct triangles can you make? ($3 \leq n \leq 1M$) ". 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. Hint: 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.

Programming Exercises related to Combinatorics (also see Fibonacci Numbers above):

1. UVa 326 - Extrapolation using a Difference Table
2. UVa 369 - Combinations (be careful with overflow issue)
3. UVa 991 - Safe Salutations (Catalan Numbers)
4. UVa 10007 - Count the Trees (Catalan Numbers, see [30] for details)
5. UVa 10219 - Find the Ways! (use Java BigInteger class)
6. UVa 10303 - How Many Trees (Catalan Numbers)
7. UVa 10375 - Choose and Divide
8. UVa 10784 - Diagonal
9. UVa 10790 - How Many Points of Intersection?

10. UVa 10918 - Tri Tiling
11. UVa 11069 - A Graph Problem
12. UVa 11115 - Uncle Jack (N^D, use Java BigInteger class)
13. UVa 11204 - Musical Instruments (only 1st choice matters)
14. UVa 11310 - Delivery Debacle
15. UVa 11401 - Triangle Counting (spot the pattern, coding is easy)
16. UVa 11554 - Hapless Hedonism (similar to UVa 11401)

5.5.2 Cycle-Finding

Cycle-finding [32] is a problem of finding a **cycle** in a sequence of **iterated function values**. For any function $f : S \to S$ and any initial value $x_0 \in S$, 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 (**cycle**), i.e. $\exists i \neq j$ such that $x_i = x_j$. Once this happens, the sequence must repeat the cycle of values from x_i to x_{j-1}. We let μ as the smallest index i and let λ (the loop length) be the smallest positive integer such that $x_\mu = x_{\mu+\lambda}$. The cycle-finding is the problem of finding μ and λ, given f 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.

A naïve algorithm that works in general for this problem uses a data structure (e.g. C++ STL <set>, hash table or direct addressing table) to store information that a number x_i has been visited in the sequence and then for each x_j found later ($j > i$), we test if x_j is stored in the data structure or not. 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.

There is a better algorithm called Floyd's Cycle-Finding algorithm [32] that runs in the same $O(\mu+\lambda)$ time complexity but only uses $O(1)$ memory space – much smaller than the naïve version. The working C/C++ implementation of this algorithm is shown below:

```
pair<int, int> floyd_cycle_finding(int (*f)(int), int x0) {
// The main phase of the algorithm, finding a repetition x_i = x_2i, hare 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)); }
// Find the position of mu, the hare and tortoise move at the same speeds
  int mu = 0; hare = tortoise; tortoise = x0;
  while (tortoise != hare) { tortoise = f(tortoise); hare = f(hare); mu += 1; }
// Find the length of the shortest cycle starting from x_mu, hare moves, tortoise stays
  int lambda = 1; hare = f(tortoise);
  while (tortoise != hare) { hare = f(hare); lambda += 1; }
  return make_pair(mu, lambda);
}
```

Programming Exercises related to Cycle-Finding:

1. UVa 350 - Pseudo-Random Numbers (straightforward application of floyd's cycle-finding)
2. UVa 408 - Uniform Generator

3. UVa 944 - Happy Numbers (similar to UVa 10591)
4. UVa 10591 - Happy Number
5. UVa 11036 - Eventually periodic sequence
6. UVa 11053 - Flavius Josephus Reloaded
7. UVa 11549 - Calculator Conundrum

5.5.3 Existing (or Fictional) Sequences and Number Systems

Some Ad Hoc mathematic problems involve a definition of existing (or fictional) number **Sequence** or **Number System** and our task is to produce either the sequence/number within some range or the n-th one, verify if the given sequence/number is valid according to definition, etc. Usually, following the description carefully is sufficient to solve the problem.

Programming Exercises related to Sequences and Number Systems:

- Sequences
  1. UVa 100 - The 3n + 1 problem
  2. UVa 413 - Up and Down Sequences
  3. UVa 694 - The Collatz Sequence (similar to UVa 100)
  4. UVa 10408 - Farey Sequences
  5. UVa 10930 - A-Sequence
  6. UVa 11063 - B2 Sequences
- Number Systems
  1. UVa 136 - Ugly Numbers
  2. UVa 138 - Street Numbers
  3. UVa 443 - Humble Numbers
  4. UVa 640 - Self Numbers (DP)
  5. UVa 962 - Taxicab Numbers (Pre-calculate the answer)
  6. UVa 974 - Kaprekar Numbers
  7. UVa 10001 - Bangla Numbers
  8. UVa 10006 - Carmichael Numbers
  9. UVa 10042 - Smith Numbers
  10. UVa 10044 - Erdos Numbers (solvable with BFS)
  11. UVa 10591 - Happy Number (solvable with the Floyd's Cycle-Finding algorithm)
  12. UVa 11461 - Square Numbers
  13. UVa 11472 - Beautiful Numbers

5.5.4 Probability Theory

Probability Theory is a branch of mathematics concerned with 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, such problems are either solve-able with some closed-form formula, or one has no choice than to enumerate the complete search space.

Programming Exercises related to Probability:

1. UVa 474 - Heads Tails Probability
2. UVa 542 - France 98
3. UVa 10056 - What is the Probability?
4. UVa 10491 - Cows and Cars
5. UVa 11176 - Winning Streak
6. UVa 11181 - Probability|Given
7. UVa 11500 - Vampires

5.5.5 Linear Algebra

A **linear equation** is defined as an equation where the order of the unknowns (variables) is **linear** (a constant or a product of a constant plus the first power of an unknown), e.g. `X + Y = 2`. **System of linear equations** is defined as a collection of n unknowns (variables) in n linear equations, e.g. `X + Y = 2` and `2X + 5Y = 6`, where the solution is `X` $= 1\frac{1}{3}$, `Y` $= \frac{2}{3}$. Notice the difference to the **linear diophantine equation** as the solution for **system of linear equations** can be non-integers!

There are several ways to find the solution for a system of linear equations. One of them is 'Gaussian Elimination'. See [5, 37] for details.

Programming Exercises related to Linear Algebra:

1. UVa 10089 - Repackaging
2. UVa 10109 - Solving Systems of Linear Equations
3. UVa 10309 - Turn the Lights Off

5.6 Chapter Notes

We admit that there are still a lot of other mathematics problems and algorithms beyond this chapter. This last section provides pointers for a few more topics.

For an even **faster prime testing** function than the one presented here, one can use the non deterministic **Miller-Rabin's** algorithm [41] – 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. For a **faster integer factorization**, one can use the **Pollard's rho** algorithm [4]. However, if the integer to be factored is a large prime number, then this is still a slow business. This fact is the security part in modern cryptography techniques.

There are other theorem, hypothesis, and conjectures about prime numbers, e.g. Carmichael's function, Riemann's hypothesis, Goldbach's conjecture, twin prime conjecture, etc. However, when such things appear in *programming* contests, usually their definitions are given!

We can compute $fib(n)$ in $O(\log n)$ using matrix multiplication, but this is usually not needed in contest setting unless the problem setter use a very big n.

Other mathematics problems that may appear in programming contests are those involving: **Chinese Remainder Theorem** (e.g. UVa 756 - Biorhythms), **Divisibility** properties (e.g. UVa 995), **Pascal's Triangle**, **Combinatorial Games** (e.g. the **Sprague-Grundy's theorem** for games like UVa 10165 - Stone Game (Nim game), Chess, Tic-Tac-Toe, etc), problems involving non-conventional **Grid** system (e.g. UVa 10182 - Bee Maja), etc.

Note that (Computational) Geometry is also part of Mathematics, but since we have a special chapter for that, we reserve the discussions related to geometry problems in Chapter 7.

In terms of doing well in ICPC, it is a good idea to have at least one strong mathematician in your ICPC team. This is because there usually exists one or two mathematics problems in the set where the solution is short but getting the solution/formula requires a strong thinking cap.

We suggest that interested readers should browse more about number theory – see books like [20], `http://mathworld.wolfram.com/`, Wikipedia and do more programming exercises related to mathematics problems, visit `http://projecteuler.net/` [7].

There are approximately **175 programming exercises** discussed in this chapter.

Chapter 6

String Processing

Human Genome has approximately 3.3 Giga base-pairs
— **Human Genome Project**

In this chapter, we present one more topic that is tested in ICPC, namely: string processing. Processing (long) string is quite common in the research field of bioinformatics and some of such problems are presented as contest problems in ICPC.

6.1 Overview and Motivation

Although not as frequent as graph and mathematics problems in the previous two chapters, string processing problems are also found in recent ICPCs (see Table 6.1). Some string-related problems have huge inputs. Thus the solution must use efficient data structures and algorithms for string.

LA	Problem Name	Source	String Problem (Algorithm)
2460	Searching Sequence ...	Singapore01	Classical String Alignment Problem (DP)
2972	A DP Problem	Tehran03	Tokenize Linear Equation
3170	AGTC	Manila06	Classical String Edit Problem (DP)
3669	String Cutting	Hanoi06	Ad Hoc String Processing
3791	Team Arrangement	Tehran06	Ad Hoc String Processing
3901	Editor	Seoul07	Longest Repeated Substring
3999	The longest constant gene	Danang07	Longest Common Substring of >= 2 strings
4200	Find the Format String	Dhaka08	Ad Hoc String Processing
4657	Top-10	Jakarta09	Suffix Array + Segment Tree

Table 6.1: Some String Processing Problems in Recent ACM ICPC Asia Regional

6.2 Ad Hoc String Processing Problems

We start this chapter by mentioning Ad Hoc string processing problems. They are contest problems involving string that require no more than basic programming skills. We only need to read the requirements in the problem description carefully and code it. Sometimes, pure character array (a.k.a. string) manipulation is sufficient. Sometimes, we need string libraries like C <string.h>,

C++ <string> class, or Java String class. For example, we can use `strstr` in C to find certain substring in a longer string (also known as string matching or string searching), `strtok` in C to tokenize longer string into tokens based on some delimiters. Here are some examples:

Programming Exercises related to Ad Hoc String Processing:

1. UVa 148 - Anagram Checker (+ backtracking)
2. UVa 159 - Word Crosses
3. UVa 263 - Number Chains
4. UVa 353 - Pesky Palindromes
5. UVa 401 - Palindromes
6. UVa 409 - Excuses, Excuses! (string matching)
7. UVa 422 - Word Search Wonder (string searching in a grid)
8. UVa 537 - Artificial Intelligence?
9. UVa 644 - Immediate Decodability
10. UVa 865 - Substitution Cypher (simple character substitution mapping)
11. UVa 902 - Password Search
12. UVa 10010 - Where's Waldorf? (string searching in a grid)
13. UVa 10115 - Automatic Editing
14. UVa 10197 - Learning Portuguese
15. UVa 10293 - Word Length and Frequency
16. UVa 10391 - Compound Words (Use efficient Data Structure!)
17. UVa 10508 - Word Morphing
18. UVa 10815 - Andy's First Dictionary
19. UVa 10878 - Decode the Tape
20. UVa 10896 - Known Plaintext Attack
21. UVa 11056 - Formula 1 (involving case insensitive string comparison)
22. UVa 11062 - Andy's Second Dictionary
23. UVa 11221 - Magic Square Palindrome (solvable without DP)
24. UVa 11233 - Deli Deli
25. UVa 11278 - One-Handed Typist
26. UVa 11362 - Phone List
27. UVa 11385 - Da Vinci Code (string manipulation + Fibonacci numbers)
28. UVa 11048 - Automatic Correction of Misspellings
29. UVa 11713 - Abstract Names (modified string comparison function)
30. UVa 11716 - Digital Fortress (simple cipher)
31. UVa 11734 - Big Number of Teams will Solve This (modified string comparison function)

However, recent contest problems in ACM ICPC usually do not ask solutions for basic string processing except for the 'giveaway' problem that all teams should be able to solve. Some string processing problems are solve-able with Dynamic Programming (DP) technique. We discuss them in Section 6.3. Some other string processing problems have to deal with **long** strings, thus an efficient data structure for string like Suffix Tree or Suffix Array must be used. We discuss these data structures and several specialized algorithms using these data structures in Section 6.4.

6.3 String Processing with Dynamic Programming

In this section, we discuss some string processing problems that solve-able with DP techniques discussed in Section 3.4.

6.3.1 String Alignment (Edit Distance)

The String Alignment (or Edit Distance) problem is defined as follows: Given two strings `A` and `B`, align[1] `A` with `B` with the maximum alignment score (or minimum number of edit operations):

After aligning `A` with `B`, there are few possibilities between character `A[i]` and `B[i]` $\forall$ index i:
1. Character `A[i]` and `B[i]` **match** (assume we give '+2' score),
2. Character `A[i]` and `B[i]` **mismatch** and we replace `A[i]` with `B[i]` ('-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:

```
A = 'information'    -> '___information_'
B = 'bioinformatics' -> 'bioinformatic_s'
                        ---222222222--- -> String Alignment Score = 9 x 2 - 6 = 12
```

A brute force solution that tries all possible alignments will typically end up with a TLE verdict for long strings `A` and/or `B`. The solution for this problem is a well-known DP solution (Needleman-Wunsch's algorithm [24]). Consider two strings `A[1 ... n]` and `B[1 ... m]`. We define $V(i, j)$ to be the score of the optimal alignment between `A[1 ... i]` and `B[1 ... j]` and $score(A, B)$ is the score if character A is aligned with character B.

Base case:
$V(0, 0) = 0$ // no score for matching two empty strings

Recurrences: For $i > 0$ and $j > 0$:
$V(0, j) = V(0, j - 1) + score(\_, B[j])$ // insert space j times to make alignment
$V(i, 0) = V(i - 1, 0) + score(A[i], \_)$ // delete i times to make alignment
$V(i, j) = max(option1, option2, option3)$, 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
$option3 = V(i, j - 1) + score(\_, B[j])$ // insert

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 re-computation of overlapping subproblems (i.e. basically a DP technique).

```
  xxx...xx      xxx...xx    xxx...x_
        |             |           |
  xxx...yy      yyy...y_    yyy...yy
match/mismatch    delete      insert
```

With a simple cost function where a match gets a +2 point and mismatch, insert, delete all get a -1 point, the detail of string alignment score of `A = 'ACAATCC'` and `B = 'AGCATGC'` is shown

[1] Align is a process of inserting spaces to strings A or B such that they have the same number of characters.

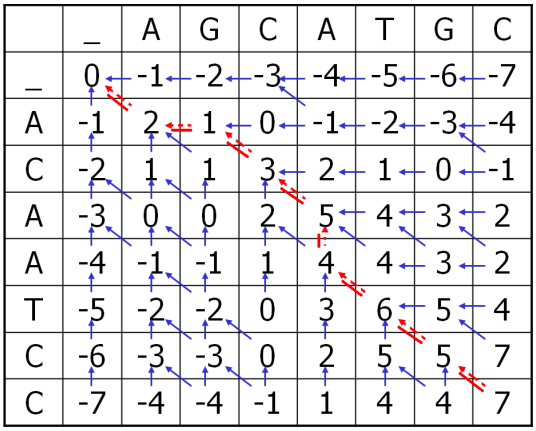


Figure 6.1: String Alignment Example for A = 'ACAATCC' and B = 'AGCATGC' (score = 7)

in Figure 6.1. The alignment score is 7 (bottom right). Follow the dashed (red) arrows from the bottom right cell to reconstruct the solution Diagonal arrow means a match or a mismatch (e.g. the last 'C'). Vertical arrow means a deletion (e.g. `..CAT..` to `..C_A..`). Horizontal arrow means an insertion (e.g. `A_C..` to `AGC..`).

```
A = 'A_CAAT[C]C'
B = 'AGC_AT[G]C'
```

As we need to fill in all entries in the table of $n \times m$ matrix and each entry can be computed in $O(1)$, the time complexity is $O(nm)$. The space complexity is $O(nm)$ – the size of the DP table.

6.3.2 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'`.

The LCS solution is very similar to String Alignment solution presented earlier. Set the cost for mismatch as negative infinity, cost for insertion and deletion as 0, and the cost for match as 1. This makes the String Alignment solution to never consider mismatches.

6.3.3 Palindrome

A palindrome is a string that can be read the same way in either direction. Some variants of palindrome finding problems are solve-able with DP technique, as shown in this example: 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 similar
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 164 - String Computer (String Alignment/Edit Distance)
2. UVa 531 - Compromise (Longest Common Subsequence + printing solution)
3. UVa 10066 - The Twin Towers (Longest Common Subsequence - but not on 'string')
4. UVa 10100 - Longest Match (Longest Common Subsequence)
5. UVa 10192 - Vacation (Longest Common Subsequence)
6. UVa 10405 - Longest Common Subsequence (as the problem name implies)
7. UVa 10739 - String to Palindrome
8. UVa 11151 - Longest Palindrome
9. LA 2460 - Searching Sequence Database in Molecular Biology (String Alignment)

6.4 Suffix Tree and Suffix Array

Suffix Tree and Suffix Array are two efficient and related data structures for strings. We do not put this topic in Chapter 2 as these two data structures are special for strings.

6.4.1 Suffix Tree: Basic Ideas

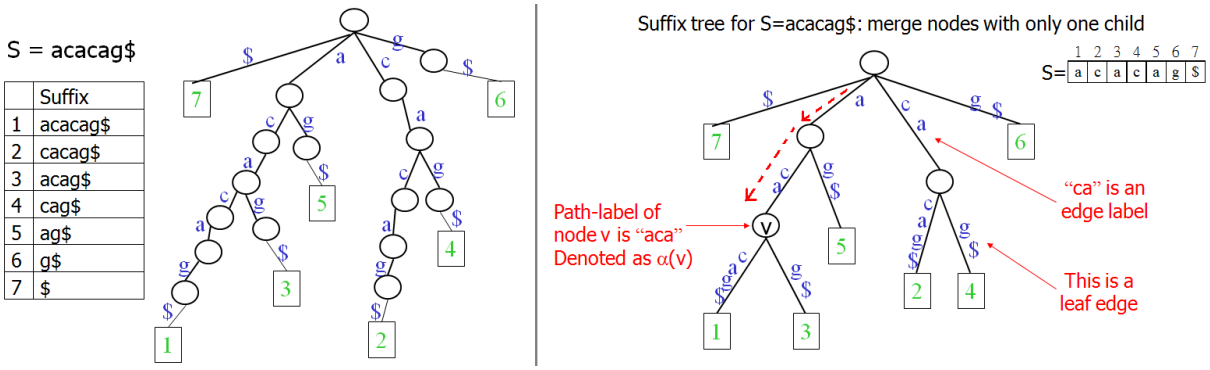


Figure 6.2: Suffix Trie (Left) and Suffix Tree (Right) of S = 'acacag$' (Figure from [24])

Consider a string[2] `S = 'acacag$'`, a Suffix[3] **Trie** of `S` is a tree that contains all possible suffixes of `S` (see Figure 6.2, left). Two suffixes that share common prefix will share the same first few vertices, e.g. `'cag$'` and `'cacag$'` share the first two vertices `'ca'` before they split. The leaves contain the indices of the suffixes. Suffix **Tree** of `S` is Suffix Trie where we merge vertices with only one child (see Figure 6.2, right). Notice the 'edge-label' and 'path-label' in the figure.

Exercise: Draw the Suffix Tree of `S = 'competitive$'`!

[2]Notice that S is usually appended with a special char '$' which is lexicographically smaller than all the alphabets used in S.

[3]A suffix of a string is a 'special case' substring that goes up to the last character of the string.

6.4.2 Applications of Suffix Tree

Assuming that a Suffix Tree[4] for a string `S` is already built, we can use it for these applications:

Exact String Matching in $O(|Q| + occ)$

With Suffix Tree, we can find all (exact) occurrences of a query string Q in S in $O(|Q|+occ)$ where $|Q|$ is the length of the query string Q itself and occ is the total number of occurrences of Q in S – no matter how long the string S is. When the Suffix Tree is already built, this approach is *faster* than many exact string matching algorithms (e.g. KMP).

With Suffix Tree, our task is to search for the vertex x in the Suffix Tree which represents the query string Q. This can be done by just one root to leaf traversal that follows the edge labels. Vertex with path-label = Q is the desired vertex x. Then, leaves in the subtree rooted at x are the occurrences of Q in S. We can then read the starting indices of such substrings that are stored in the leaves of the sub tree.

For example, in the Suffix Tree of `S = 'acacag$'` shown in Figure 6.2, right and `Q = 'aca'`, we can simply traverse from root, go along the edge label 'a', then the edge label 'ca' to find vertex x with the path-label 'aca' (follow the dashed red arrow in Figure 6.2, right). The leaves of this vertex x point to index 1 (substring: `'acacag$'`) and index 3 (substring: `'acag$'`).

Exercise: Now try to find a query string `Q = 'ca'` and `Q = 'cat'`!

Finding Longest Repeated Substring in $O(n)$

With Suffix Tree, we can also find the longest repeated substring in S easily. The deepest internal vertex X in the Suffix Tree of S 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 represent more than one suffixes (leaves) of string S and these suffixes shared a common prefix (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 `S = 'acacag$'` shown in Figure 6.2, right, the longest repeated substring is 'aca' as it is the path-label of the deepest internal vertex.

Exercise: Find the longest repeated substring in `S = 'cgacattacatta$'`!

Finding Longest Common Substring in $O(n)$

The problem of finding the Longest Common **Substring** (not Subsequence)[5] of two **or more** strings can be solved in linear time with Suffix Tree. Consider two strings $S1$ and $S2$, we can build a **generalized Suffix Tree** for $S1$ and $S2$ with two different ending markers, e.g. $S1$ with character '#' and $S2$ with character '$'. Then, we mark each internal vertices with have leaves that represent suffixes of *both* $S1$ and $S2$ – this means the suffixes share a common prefix. We then report the deepest marked vertex as the answer.

For example, with `S1 = 'acgat#'` and `S2 = 'cgt$'`, The Longest Common Substring is `'cg'` of length 2. In Figure 6.3, we see the root and vertices with path-labels 'cg', 'g', and 't' all have two different leaf markers. The deepest marked vertex is 'cg'. The two suffixes `cgat#` and `cgt$` share a common prefix 'cg'.

[4] As Suffix **Tree** is more compact than Suffix **Trie**, we will concentrate on Suffix **Tree**.

[5] In 'abcdef', 'bce' (skip character 'd') is subsequence and 'bcd' (contiguous) is substring and also subsequence.

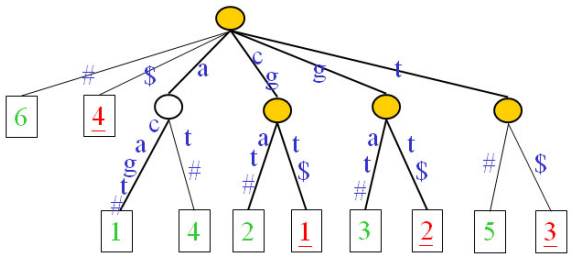


Figure 6.3: Generalized Suffix Tree of S1 = 'acgat#' and S2 = 'cgt$' (Figure from [24])

Exercise: Find the Longest Common Substring of `S1 = 'steven$'` and `S2 = 'seven$'`!

6.4.3 Suffix Array: Basic Ideas

Building efficient Suffix Tree under contest environment is a bit complex and risky. Fortunately, another data structure that we are going to describe, called Suffix **Array** invented by Udi Manber and Gene Myers, has similar functionalities as Suffix Tree but simpler to implement, especially in programming contest setting. Thus we will skip the discussion on building a Suffix Tree and instead focus on Suffix Array.

Basically, Suffix Array is an integer array that contains indices of sorted suffixes. For example, consider the same `S = 'acacag$'` with $n = 7$. The Suffix Array of S is an integer permutation of n as shown in Figure 6.4. Note that we regard the terminating symbol `$` as the lexicographically smallest character.

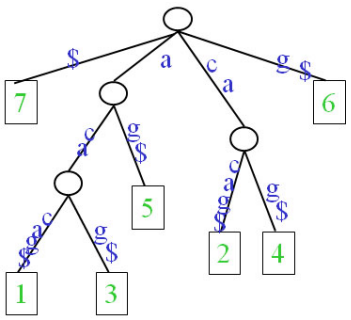
Suffix	Position
acacag$	1
cacag$	2
acag$	3
cag$	4
ag$	5
g$	6
$	7

=> **Sort**

SA[i]	Suffix
7	$
1	acacag$
3	acag$
5	ag$
2	cacag$
4	cag$
6	g$

Figure 6.4: Suffix Array of S = 'acacag$' (Figure from [24])

Suffix Tree and Suffix Array are very related. As we can see in Figure 6.5, the leaves of a Suffix Tree (from left to right) is in Suffix Array order. In short, a **vertex** in Suffix Tree corresponds to a **range** in Suffix Array!



SA[i]	Suffix
7	$
1	acacag$
3	acag$
5	ag$
2	cacag$
4	cag$
6	g$

Figure 6.5: Suffix Tree versus Suffix Array of S = 'acacag$' (Figure from [24])

A Suffix Array is good enough for many practical string operations in contest problems. In this section, we present two simple ways to build a Suffix Array given a string `S[0 ... n-1]`.

```
#include <iostream>
#include <stdlib.h>
#include <string.h>
using namespace std;

char S[1001]; // this naive Suffix Array cannot go beyond 1000 characters
int SA[1001], n;                          // compare suffixes
int SA_cmp(const void *a, const void *b) { return strcmp(S + *(int*)a, S + *(int*)b); }

int main() { // first approach: O(n^2 log n), only for n <= 1000
  n = strlen(gets(S));
  for (int i = 0; i < n; i++) SA[i] = i; // sort       * comparison
  qsort(SA, n, sizeof(int), SA_cmp);     // O(n log n) * O(n) = O(n^2 log n)
  for (int i = 0; i < n; i++) printf("%d %s\n", SA[i], S+SA[i]);
} // return 0;
```

When applied to string `S = 'acacag$'`, the simple code that simply sort all suffixes with sort library will produce the correct Suffix Array = {6, 0, 2, 4, 1, 3, 5} (note that index starts from 0). However, this is barely useful except for contest problems with $n \leq 1000$.

A better way to build the Suffix Array is to sort suffixes $O(n \log n)$ in increasing length. We start from suffixes with length 1, length 2, length 4, length 8, ..., up to n. As the length grows exponentially, we only need $O(\log n)$ steps. Thus the overall complexity is $O(n \log^2 n)$. With this complexity, working with strings of length $n \leq 100K$ – the typical programming contest range – is not a problem. The library code is shown below. For explanation, see [29, 24].

```
#include <algorithm>
#include <iostream>
#include <stdlib.h>
#include <string.h>
using namespace std;
#define MAXN 200010
int RA[MAXN], SA[MAXN], LCP[MAXN], *FC, *SC, step;
char S[MAXN], Q[MAXN];

bool cmp(int a, int b) {
  if (step==-1 || FC[a]!=FC[b]) return FC[a] < FC[b];
  return FC[a+(1<<step)] < FC[b+(1<<step)];
}

void suffix_array(char *S, int n) { // O(n log^2(n))
  for (int i=0; i<n; i++) RA[i] = S[SA[i] = i];
  for (FC=RA, SC=LCP, step=-1; (1<<step)<n; step++) {
    sort(SA, SA+n, cmp);
    int cnt = 0;
    for (int i=0; i<n; i++) {
      if (i>0 && cmp(SA[i-1],SA[i])) cnt++;
      SC[SA[i]] = cnt;
    }
    if (cnt==n-1) break; // all distinct, no need to continue
    swap(FC,SC);
  }
  for (int i=0; i<n; i++) RA[SA[i]] = i;
}
```

Now with a Suffix Array already built, we can search for a query string Q of length m in string S of length n in $O(m \log n)$. This is $O(\log n)$ 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 $O(\log n)$ binary search on a sorted suffixes and do up to $O(m)$ comparisons per suffix.

The fact that the occurrences of Q in the Suffix Array of S are consecutive can be used to deal with the longest repeated substring and the longest common substring problems in similar $O(m \log n) + O(occ)$. For example in `S = 'acacag$'`, the repeated substring `'ca'` occurs in SA[4] and SA[5]. For the common substring between `S1 = 'acgat#'` and `S2 = 'cgt$'`, we can concatenate the two string into `S = 'acgat#cgt$'`, build the Suffix Array of the concatenated string, and then modify the comparison function to check for the marker character '#' and '$'.

Our code for finding the query string Q in a Suffix Array is shown below:

```
pair<int, int> range(int n, char *Q) {
  int lo = 1, hi = n, m = strlen(Q), mid = lo; // index 0 - null
  while (lo <= hi) {                           // valid range = [1..n]
    mid = (lo + hi) / 2;
    int cmp = strncmp(S + SA[mid], Q, m);
    if (cmp == 0)     break; // found
    else if (cmp > 0) hi = mid - 1;
    else              lo = mid + 1;
  }

  if (lo > hi)
    return make_pair(-1, -1); // not found

  for (lo = mid; lo >= 1 && strncmp(S + SA[lo], Q, m) == 0; lo--);
  lo++;
  for (hi = mid; hi <= n && strncmp(S + SA[hi], Q, m) == 0; hi++);
  hi--;

  return make_pair(lo, hi);
}

int main() {
  int n = strlen(gets(S));
  suffix_array(S, n + 1); // NULL is included!
  for (int i = 1; i <= n; i++) // SA[0] is the NULL
    printf("%d %s\n", SA[i], S + SA[i]);

  gets(Q);
  pair<int, int> pos = range(n, Q);
  if (pos.first != -1 && pos.second != -1) {
    printf("%s is found SA [%d .. %d] of %s\n", Q, pos.first, pos.second, S);
    printf("They are:\n");
    for (int i = pos.first; i <= pos.second; i++)
      printf("  %s\n", S + SA[i]);
  }
  else
    printf("%s is not found in %s\n", Q, S);

  return 0;
}
```

Programming Exercises related to Suffix Array:

1. UVa 719 - Glass Beads (minimum lexicographic rotation, solvable with SA)
2. UVa 10526 - Intellectual Property
3. UVa 11107 - Life Forms
4. UVa 11512 - GATTACA
5. LA 4657 - Top 10 (ACM ICPC Jakarta 2009, problem setter: Felix Halim)
6. https://www.spoj.pl/problems/SARRAY/ (problem setter: Felix Halim)

6.5 Chapter Notes

The materials regarding String Alignment (Edit), Longest Common Subsequence, Suffix Tree, and Suffix Array are courtesy of **A/P Sung Wing Kin, Ken** [24] from School of Computing, National University of Singapore.

The string alignment problem discussed in Section 6.3 is called the **global** alignment problem. The best solution so far is $O(nm/\log^2 n)$ [24] but it is complicated to code. However if the given contest problem is limited to d insertions or deletions only, we can speed up Needleman-Wunsch's algorithm to $O(dn)$ by just concentrating on the main diagonal of DP matrix.

For the specialized case of **local** alignment problem, a better algorithm similar to Needleman-Wunsch's algorithm exists (**Smith-Waterman's** algorithm [24]).

The Longest Common Subsequence problem can be solved in $O(n \log n)$ when all characters are distinct [24].

There are other string matching algorithms that are not discussed in this chapter: **Rabin-Karp's**, **Boyer-Moore's**, **Knuth-Morris-Pratt's (KMP)**, **Aho-Corasick's**, etc. But for many contest problems involving string, using library functions like C `strstr` in <`string.h`> or `string.find()` in C++ STL <`string`> is usually already sufficient. If a faster string matching algorithm is needed during contest time, we suggest using Suffix Array that has been discussed in this chapter.

For more example applications of Suffix Array, please read the article "Suffix arrays - a programming contest approach" by [29].

There are several other string processing problems that we have not touched yet: **Grammar and Parsing** (e.g. Backus Naur Form), **Encoding/Decoding/Cypher/Cryptography**, **Shortest Common Superstring**, **Regular Expression** (using Java), **Hashing techniques** for String Processing, **Suffix Automaton**, etc.

There are approximately **54 programming exercises** discussed in this chapter.

Chapter 7

(Computational) Geometry

Let no man ignorant of geometry enter here.
— **Plato's Academy in Athens**

(Computational) Geometry is yet another topic that frequently appears in programming contests. Many contestants afraid to tackle them due to floating point precision errors[1] *or the many tricky 'special cases' commonly found in geometry problems. Some others skip these problems as they forgot some important formulas and unable to derive the required formulas from basic concepts. Study this chapter for some ideas on tackling (computational) geometry problems in ICPC.*

7.1 Overview and Motivation

Almost all ICPC problem sets have at least one geometry problem. If you are lucky, it ask you for some geometry solution that you have learned in pre-University which you still remember or the solution can be derived by drawing the geometrical objects. However, many geometry problems are more complex and finding the correct algorithm is harder. Table 7.1 shows some geometry problems in recent ACM ICPC Asia Regional.

LA	Problem Name	Source	Geometry Problem (Algorithm)
2797	Monster Trap	Aizu03	
3169	Boundary Points	Manila06	Convex Hull
3616	How I Wonder ...	Yokohama06	
4107	TUSK	Singapore07	
4410	Shooting the Monster	KLumpur08	
4413	Triangle Hazard	KLumpur08	
4532	Magic Rope	Hsinchu09	
4639	Separate Points	Tokyo09	
4642	Malfatti Circles	Tokyo09	Triangles & Circles
4717	In-circles Again	Phuket09	Circles in Triangle

Table 7.1: Some (Computational) Geometry Problems in Recent ACM ICPC Asia Regional

[1]To avoid this error, usually we do floating-point comparison test in this way: $fabs(a - b) < EPS$ where EPS usually is a small number like 1e-9.

We divide this chapter into two parts. The first part is **geometry basics** in Section 7.2. We review many (not all) English geometric terminologies and formulas that are commonly used in programming contests. The second part deals with **computational geometry** in Section 7.4 - 7.5, where we use *data structures and algorithms* which can be stated in terms of geometry.

7.2 Geometry Basics

As ACM ICPC contestants come from various nationalities, languages, and backgrounds, sometimes these English geometric terminologies below may look alien. Here, we try to provide a list – which cannot be exhaustive – to be used as a quick reference when contestants are given geometry problems.

- Lines

  1. A **Line** can be described with mathematical equation: $y = mx + c$ or $ax + bx + c = 0$. The $y = mx + c$ equation involves 'gradient' / 'slope' m.
     Note: Be careful with vertical lines with 'infinite' slope. Usually, we treat the vertical lines separately in the solution code (example of the *special cases* in geometry problems).
  2. A Line **Segment** is a line with two end points with finite length.

Programming Exercises related to Lines:

1. UVa 184 - Laser Lines
2. UVa 270 - Lining Up
3. UVa 833 - Water Falls
4. UVa 10180 - Rope Crisis In Ropeland (closest point on line segment)
5. UVa 10242 - Fourth Point
6. UVa 10263 - Railway
7. UVa 11068 - An Easy Task (line ax + by + c)
8. UVa 11277 - The Silver Bullet (Complete Search + collinear test discussed below)
9. LA 4601 - Euclid (Southeast USA Regional 2009)

- Circles (see Figure 7.1)

  1. In a 2-D *Cartesian* coordinate system, the **Circle** centered at (a, b) with radius r is the set of all points (x, y) such that $(x - a)^2 + (y - b)^2 = r^2$.
  2. The constant π is the **Ratio** of *any* circle's circumference to its diameter in the Euclidean space. To avoid precision error, the safest value for programming contest is `pi` $= 2 \times acos(0.0)$, unless if this constant is defined in the problem description!
  3. The **Circumference** c of a circle with a **Diameter** d is $c = \pi \times d$ where $d = 2 \times r$.
  4. The length of an **Arc** of a circle with a circumference c and an angle α (in degrees[2]) is $\frac{\alpha}{360.0} \times c$

[2]Human usually works with degrees, but many mathematical functions in programming languages works with radians. Check your programming language manual to verify this. To help with conversion, just remember that one π radian equals to 180 degrees.

5. The length of a **Chord** of a circle with a radius r and an angle α (in degrees) can be obtained with the **Law of Cosines**: $2r^2 \times (1 - cos(\alpha))$ – see the explanation of this law in the discussion about Triangles below.
6. The **Area** A of a circle with a radius r is $A = \pi \times r^2$
7. The area of a **Sector** of a circle with an area A and an angle α (in degrees) is $\frac{\alpha}{360.0} \times A$
8. The area of a **Segment** of a circle can be found by subtracting the area of the corresponding Sector with the area of an Isosceles Triangle with sides: r, r, and Chord-length.

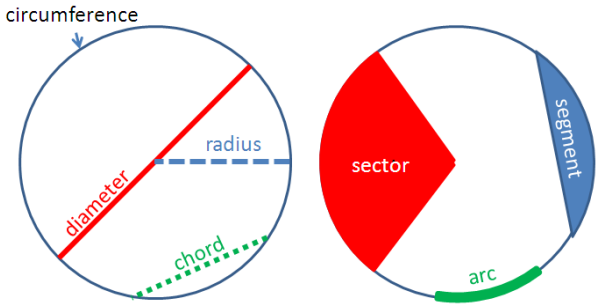


Figure 7.1: Circles

Programming Exercises related to Circles (only):

1. UVa 10005 - Packing polygons (Complete Search: Circle through 2 points, test others)
2. UVa 10012 - How Big Is It (Circle radius, use complete search)
3. UVa 10136 - Chocolate Chip Cookies (Circle through 2 points)
4. UVa 10221 - Satellites (finding arc and chord length of a circle)
5. UVa 10432 - Polygon Inside A Circle (area of n-sided regular polygon inside circle)
6. UVa 10451 - Ancient Village Sports (inner/outer circle of an n-sided regular polygon)
7. UVa 10589 - Area (check if point is inside intersection of 4 circles)

- Triangles (see Figure 7.2)

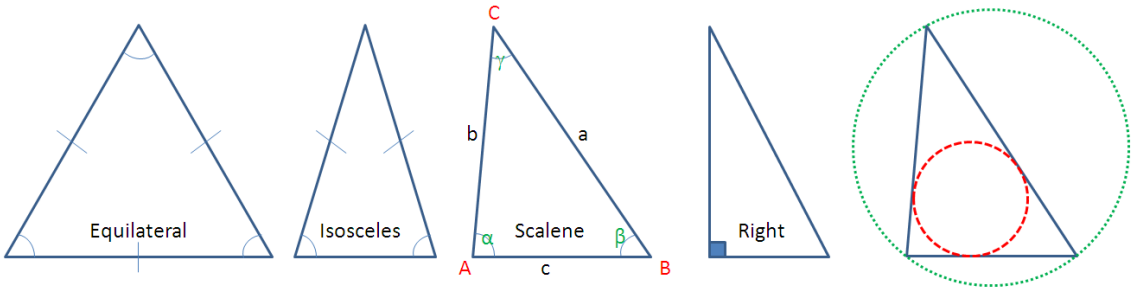


Figure 7.2: Triangles

1. A **Triangle** is a polygon (defined below) with three vertices and three edges. There are several types of triangles:
   Equilateral Triangle, all three edges have the same length and all inside/interior angles are 60 degrees;

Isosceles Triangle, two edges have the same length;
Scalene Triangle, no edges have the same length;
Right Triangle, one of its interior angle is 90 degrees (or a **right angle**).

2. The **Area** A of triangle with base b and height h is $A = 0.5 \times b \times h$
3. The **Perimeter** p of a triangle with three sides: a, b, and c is $p = a + b + c$.
4. The **Heron's Formula**: The area A of a triangle with 3 sides: a, b, c, is $A = sqrt(s \times (s-a) \times (s-b) \times (s-c))$, where $s = 0.5 \times p$ (the **Semi-Perimeter** of the triangle).
5. The radius r of the Triangle's **Inner Circle** with area A and the semi-perimeter s is $r = A/s$.
6. The radius R of the Triangle's **Outer Circle** with 3 sides: a, b, c and area A is $R = a \times b \times c/(4 \times A)$.
7. 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 7.2. With the notation described there, we have: $c^2 = a^2 + b^2 - 2 \times a \times b \times cos(\gamma)$. This formula can be rewritten for the other two angles α and β.
8. 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 7.2. With the notation described there, we have: $\frac{a}{\sin(\alpha)} = \frac{b}{\sin(\beta)} = \frac{c}{\sin(\gamma)}$.
9. In Trigonometry, the **Pythagorean Theorem** specializes the Law of Cosines. The Pythagorean theorem is only correct for 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: $c^2 = a^2 + b^2$.
10. In Trigonometry, 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. A **Primitive Pythagorean Triple** is one in which a, b, and c are coprime. Primitive Pythagorean Triples describe the three integer side lengths of a Right Triangle, although the converse may not be true.

Programming Exercises related to Triangles (and possibly Circles again):

1. UVa 143 - Orchard Trees (counting integer points in triangle)
2. UVa 190 - Circle Through Three Points (triangle's outer circle)
3. UVa 438 - The Circumference of the Circle (triangle's outer circle)
4. UVa 10195 - The Knights Of The Round Table (triangle's inner circle, Heron's formula)
5. UVa 10286 - The Trouble with a Pentagon (Law of Sines)
6. UVa 10347 - Medians (given 3 medians of a triangle, find its area)
7. UVa 10991 - Region (Heron's formula, Law of Cosines, area of sector)
8. UVa 11152 - Colourful Flowers (triangle's inner and outer circle, Heron's formula)
9. UVa 11437 - Triangle Fun

10. UVa 11479 - Is this the easiest problem? (property check)
11. UVa 11524 - In-Circle (yet another variant of circles in triangles)
12. UVa 11579 - Triangle Trouble
13. LA 4413 - Triangle Hazard
14. LA 4717 - In-circles Again

- Rectangles

1. A **Rectangle** is a polygon with four edges, four vertices, and four right angles.
2. The **Area** A of a rectangle with width w and height h is $A = w \times h$.
3. The **Perimeter** p of a rectangle with width w and height h is $p = 2 \times (w + h)$.
4. A **Square** is a special case of rectangle where $w = h$.

Programming Exercises related to Rectangles:

1. UVa 201 - Square
2. UVa 476 - Point in Figures: Rectangles
3. UVa 922 - Rectangles by the Ocean
4. UVa 10502 - Counting Rectangles (Complete Search)
5. UVa 10908 - Largest Square (implicit square)
6. UVa 11207 - The Easiest Way (cutting rectangle into 4-equal-sized squares)
7. UVa 11455 - Behold My Quadrangle (property check)

- Trapeziums

1. A **Trapezium** is a polygon with four edges, four vertices, and one pair of parallel edges. If the two non-parallel sides of the trapezium have the same length, we have an **Isosceles Trapezium**.
2. The Area A of a trapezium with base $w1$, another edge parallel with the base $w2$ and height h is $A = 0.5 \times (w1 + w2) \times h$.

- Quadrilaterals

1. A **Quadrilateral** or **Quadrangle** is a polygon with with four edges (and four vertices). Rectangles, Squares, and Trapeziums that are mentioned above are Quadrilaterals. Figure 7.3 shows a few more examples: **Parallelogram**, **Kite**, **Rhombus**.

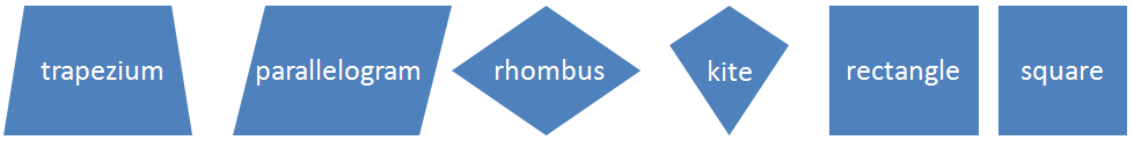


Figure 7.3: Quadrilaterals

- Spheres

1. A **Sphere** is a perfectly round geometrical object in 3-D space.
2. The **Great-Circle Distance** between any two points A and B on sphere [38] is the shortest distance along a path on the **surface of the sphere**. This path is equal to the length of the Arc of the **Great-Circle** of that sphere that pass through the two points A and B.

   The Great-Circle of a sphere is defined as the circle that runs along the surface of the sphere so that it cuts the sphere into two equal hemispheres. To find the solution to the Great-Circle Distance, we find the central angle AOB (see Figure 7.4) of the Great-Circle and use it to determine the length of the arc.

   Although quite rare, some contest problems use this distance metric. Usually, the two points on the surface of a sphere are given as (latitude, longitude) pair. The following library code will help us obtain the shortest great-circle distance given two points on the sphere and the radius of the sphere. For the derivation of this library code, see [38].

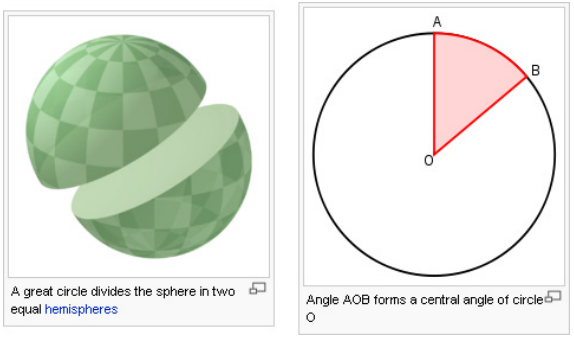


Figure 7.4: Great-Circle and Great-Circle Distance (Arc A-B) (Figures from [46])

```
double greatCircleDistance(double pLat, double pLong,
                           double qLat, double qLong, double radius) {
  pLat *= PI / 180;  pLong *= PI / 180;
  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));
}
```

Programming Exercises related to the Great-Circle Distance:

1. UVa 535 - Globetrotter
2. UVa 10075 - Airlines (with APSP)
3. UVa 10316 - Airline Hub
4. UVa 10897 - Travelling Distance
5. UVa 11817 - Tunnelling The Earth

- Polygons

1. A **Polygon** is a plane figure that is bounded by a closed path or circuit composed of a finite sequence of straight line segments. These segments are called edges or sides. The point where two edges meet are the polygon's vertex or corner. The interior of the polygon is sometimes called its body.
2. 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**.
3. The area A of an n-sided polygon (either convex or concave) with n pairs of vertex coordinates given in some order (clockwise or counter-clockwise) is:

```
          | x1  y1 |
          | x2  y2 |
     1    | x3  y3 |
A  = - * | .   .  | = 1/2 * (x1y2 + x2y3 + y3y4 + ... + xny1
     2    | .   .  |          -x2y1 - x3y2 - x3y4 - ... - x1yn)
          | .   .  |
          | xn  yn |
```

   This can be written into the library code below. Notice that our default setting is all-integer coordinates and use all-integer operations whenever possible. You need to change a part of this code if the given points are not integers:

```
struct point { int x, y; } // a point has two members

int determinant(vector<point> P) { // default: integer computation
  int result = 0, x1, y1, x2, y2;
  for (int i = 0; i < P.size(); i++) {
    x1 = P[i].x; x2 = P[(i + 1) % P.size()].x;
    y1 = P[i].y; y2 = P[(i + 1) % P.size()].y;
    result += (x1 * y2 - x2 * y1);
  }
  return result;
}

// area is half of the determinant and the result may be a non-integer
double area(vector<point> P) { return fabs(determinant(P)) / 2.0); }
```

4. The perimeter p of an n-sided polygon with n pairs of coordinates given in some order (clockwise or counter-clockwise) can be computed with Pythagorean theorem:

```
double perimeter(vector<point> P) { // point has x & y
  double result = 0.0, x1, y1, x2, y2, dx, dy;
  for (int i = 0; i < P.size(); i++) {
    x1 = P[i].x; x2 = P[(i + 1) % P.size()].x;
    y1 = P[i].y; y2 = P[(i + 1) % P.size()].y;
    dx = x2 - x1;
    dy = y2 - y1;
    result += sqrt(dx * dx + dy * dy);
  }
  return result;
}
```

5. Testing if a polygon is convex (or concave) is easy with a quite robust[3] geometric predicate test called **CCW (Counter Clockwise) Test** (a.k.a. **Left-Turn Test**).

   CCW test is a simple yet important predicate test in computational geometry. This test takes in 3 points p, q, r in a plane and determine if the sequence $p \rightarrow q \rightarrow r$ is a left turn[4]. For example, $CCW(p, q, r)$ where $p = (0, 0)$, $q = (1, 0)$, $r = (0, 1)$ is true. This test can be implemented with the following library code:

```
int turn(point p, point q, point r) {
  int result = (r.x - q.x) * (p.y - q.y) - (r.y - q.y) * (p.x - q.x);
  if (result < 0) return -1; // P->Q->R is a right turn
  if (result > 0) return  1; // P->Q->R is a left turn
  return 0; // P->Q->R is a straight line, i.e. P, Q, R are collinear
}

// Note: sometimes, we change the '> 0' to '>= 0' to accept collinear points
bool ccw(point p, point q, point r) { return (turn(p, q, r) > 0); }
```

   With the library code above, we can now check if a polygon is convex by verifying if all three consecutive points in the polygon make left-turns if visited in counter clockwise order. If we can find at least one triple where this is false, the polygon is concave.

Programming Exercises related to Polygons:

1. UVa 478 - Point in Figures: Rectangles, Circles, and Triangles (point inside polygon)
2. UVa 634 - Polygon (point inside polygon)
3. UVa 10078 - Art Gallery (testing if a polygon is convex)
4. UVa 10112 - Myacm Triangles (check if point in triangle with area tests like UVa 478)
5. UVa 11447 - Reservoir Logs (area of polygon)
6. UVa 11473 - Campus Roads (perimeter of polygon)

- There are of course exists many other geometric shapes, objects, and formulas that have not been covered yet, like 3-D objects, etc. What we have covered so far are the ones which appear more frequently in programming contests.

Other Programming Exercises related to Basic Geometry that are not listed above:

1. UVa 10088 - Trees on My Island (Georg A. Pick's Theorem: $A = i + \frac{b}{2} - 1$, see [43])
2. UVa 10297 - Beavergnaw (cones, cylinders, volumes)
3. UVa 10387 - Billiard (expanding surface, trigonometry)
4. UVa 11232 - Cylinder (circles, rectangles, cylinders)
5. UVa 11507 - Bender B. Rodriguez Problem (simulation)

---

[3]Geometric programs are *preferred* to be robust, namely, no numerical errors. To help achieving that quality, computations are often done by *predicate tests* (e.g. the CCW test) rather than by floating point calculations which is prone to precision errors. Moreover, arithmetic operations used are limited to additions, subtractions and multiplications on integers only (exact arithmetics).

[4]Or in other words: $p \rightarrow q \rightarrow r$ is counter-clockwise, r is on the left of line pq, triangle pqr has a positive area and its determinant is greater than zero.

7.3 Graham's Scan

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 (see Figure 7.5.D). Every vertex in $CH(P)$ is a vertex in original P. Thus, the algorithm of finding convex hull must decide which vertices in P to be chosen as part of the convex hull.

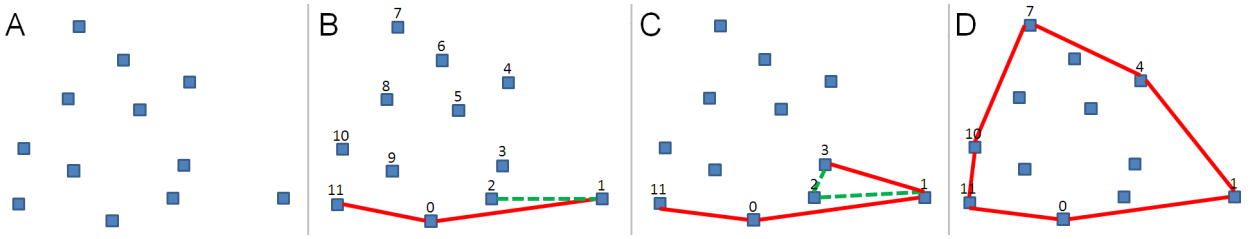


Figure 7.5: Convex Hull $CH(P)$ of Set of Points P

There are several convex hull algorithms available. In this section, we choose the $O(n \log n)$ Ronald *Graham's Scan* algorithm. This algorithm first sorts all n points of P (Figure 7.5.A) based on its angle w.r.t a point called pivot. In our example, we pick bottommost and rightmost point in P as pivot (see point 0 and the counter-clockwise order of the remaining points in Figure 7.5.B).

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. Examine Figure 7.5.C. The stack previously contains (bottom) 11-0-1-2 (top), but when we try to insert 3, 1-2-3 is a right turn, so we pop 2. Now 0-1-3 is a left turn, so we insert 3 to the stack, which now contains (bottom) 11-0-1-3 (top).

When Graham's Scan terminates, whatever left in S are the points of $CH(P)$ (see Figure 7.5.D, the stack contains (bottom) 11-0-1-4-7-10-11 (top)). Graham Scan's eliminates all the right turns! Check that the sequence of vertices in S always makes left turns – a convex polygon.

The implementation of Graham's Scan, omitting parts that have shown earlier like `ccw` function, is shown below.

```
point pivot; // global variable
vector<point> polygon, CH;

int area2(point a, point b, point c) {
  return a.x * b.y - a.y * b.x + b.x * c.y - b.y * c.x + c.x * a.y - c.y * a.x;
}

int dist2(point a, point b) { // function to compute distance between 2 points
  int dx = a.x - b.x, dy = a.y - b.y; return dx * dx + dy * dy;
}

bool angle_cmp(point a, point b) { // important angle-sorting function
  if (area2(pivot, a, b) == 0) // collinear
    return dist2(pivot, a) < dist2(pivot, b); // which one closer

  int d1x = a.x - pivot.x, d1y = a.y - pivot.y;
  int d2x = b.x - pivot.x, d2y = b.y - pivot.y;
  return (atan2((double)d1y, (double)d1x) - atan2((double)d2y, (double)d2x)) < 0;
}
```

```
vector<point> GrahamScan(vector<point> Polygon) {
  // first, find P0 = point with lowest Y and if tie: rightmost X
  int i, P0 = 0, N = Polygon.size();
  for (i = 1; i < N; i++)
    if (Polygon[i].y  < Polygon[P0].y ||
       (Polygon[i].y == Polygon[P0].y && Polygon[i].x > Polygon[P0].x))
      P0 = i;

  point temp = Polygon[0]; // swap selected vertex with Polygon[0]
  Polygon[0] = Polygon[P0];
  Polygon[P0] = temp;

  // second, sort points by angle w.r.t. P0, skipping Polygon [0]
  pivot = Polygon[0]; // use this global variable as reference
  sort(++Polygon.begin(), Polygon.end(), angle_cmp);

  // third, the ccw tests
  stack<point> S;
  point prev, now;
  S.push(Polygon[N - 1]); // put two starting vertices into stack S
  S.push(Polygon[0]);

  i = 1; // and start checking 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); // trick to get the 2nd item from top of S
    if (ccw(prev, now, Polygon[i])) { // if these 3 points make a left turn
      S.push(Polygon[i]); // accept
      i++;
    }
    else // otherwise
      S.pop(); // pop this point until we have a left turn
  }

  vector<point> ConvexHull;
  while (!S.empty()) { // from stack back to vector
    ConvexHull.push_back(S.top());
    S.pop();
  }
  ConvexHull.pop_back(); // the last one is a duplicate of first one

  return ConvexHull; // return the result
}
```

Programming Exercises related to Convex Hull:

1. UVa 109 - Scud Busters
2. UVa 218 - Moth Eradication
3. UVa 361 - Cops and Robbers
4. UVa 681 - Convex Hull Finding (pure convex hull problem for starting point)
5. UVa 811 - The Fortified Forest (Complete Search with convex hull)

6. UVa 10002 - Center of Masses (centroid of convex polygon, convex hull)
7. UVa 10065 - Useless Tile Packers (plus area of polygon)
8. UVa 10135 - Herding Frosh
9. UVa 10173 - Smallest Bounding Rectangle
10. UVa 11626 - Convex Hull

7.4 Intersection Problems

In geometry problems, we can virtually take any pair of objects and see if they intersect, and if they are: what is the point/area/volume of intersection? This is the source of many computational geometry problems in contests.

Line segment intersection is one the most frequent intersection problems. We can test whether two line segments intersect or not by using several `ccw` tests (code is shown below).

```
struct line { point p1, p2; }

int intersect(line line1, line line2) {
  return
   ((ccw(line1.p1, line1.p2, line2.p1) * ccw(line1.p1, line1.p2, line2.p2)) <= 0)
     &&
   ((ccw(line2.p1, line2.p2, line1.p1) * ccw(line2.p1, line2.p2, line1.p2)) <= 0);
}
```

However, intersections can occur between different types of objects other than two line segments. The other objects are: Cube, Box, Circle, Polygon, Triangle, Rectangle, etc. Some examples are shown below.

Programming Exercises solvable related to intersection problem:

- Line Segment Intersection
  1. UVa 191 - Intersection
  2. UVa 378 - Intersecting Lines
  3. UVa 866 - Intersecting line segments
  4. UVa 920 - Sunny Mountain
  5. UVa 972 - Horizon Line
  6. UVa 10902 - Pick-up sticks
  7. UVa 11343 - Isolated Segments
- Other Objects
  1. UVa 453 - Intersecting Circles (circle and circle)
  2. UVa 460 - Overlapping Rectangles (rectangle and rectangle)
  3. UVa 737- Gleaming the Cubes (cube and cube)
  4. UVa 904 - Overlapping Air Traffic Control (3D-box and 3D-box)
  5. UVa 10301 - Rings and Glue (circle and circle)
  6. UVa 10321 - Polygon Intersection (convex polygon and convex polygon)

7. UVa 11122 - Tri Tri (triangle and triangle)
8. UVa 11345 - Rectangles (rectangle and rectangle)
9. UVa 11515 - Cranes (circle and circle, plus DP)
10. UVa 11601 - Avoiding Overlaps (rectangle and rectangle)
11. UVa 11639 - Guard the Land (rectangle and rectangle)

7.5 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 3.2. One of the example is shown below.

Bisection Method for Geometry Problem

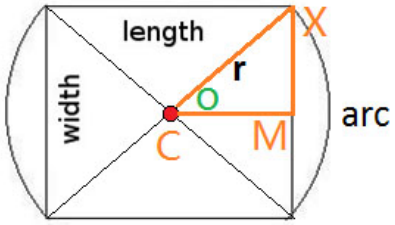


Figure 7.6: Athletics Track (from UVa 11646)

Suppose that we have a problem as follows: Given a desired soccer field with desired length : width ratio = $A : B$, two arcs from the same circle whose center is located in the middle of the field, and the length of the athletics track (the perimeter: 2× length + 2× arc) to be 400m, what is the actual length and width of the field? See Figure 7.6.

It is quite hard to find the solution with pen and paper, but with the help of computer and bisection method (binary search), we can find the solution easily.

Suppose we binary search the value of L, then we can get W from $b \times L/a$. 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 7.6). 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 enlarge or reduce the length L. The important portion of the code is shown below.

```
lo = 0.0; hi = 400.0; // the range of answer
while (fabs(lo - hi) > 1e-9) { // do bisection method on L
  L = (lo + hi) / 2.0;
  W = b * L / a; // W can be derived by W
  expected_arc = (400 - 2.0 * L) / 2.0;
  half_L = 0.5 * L; half_W = 0.5 * W;
  r = sqrt(half_L * half_L + half_W * half_W);
  angle = 2.0 * atan(half_W / half_L) * 180.0 / PI;
  this_arc = angle / 360.0 * PI * (2.0 * r);
  if (this_arc > expected_arc) hi = L;
  else                         lo = L;
}
printf("Case %d: %.12lf %.12lf\n", caseNo++, L, W);
```

Programming Exercises related to this section:

1. UVa 10245 - The Closest Pair Problem (as the problem name implies)
2. UVa 10566 - Crossed Ladders (Bisection Method)
3. UVa 11378 - Bey Battle (also a Closest Pair Problem)
4. UVa 11646 - Athletics Track (Bisection Method, the circle is at the center of track)
5. UVa 11648 - Divide The Land (Bisection Method)

7.6 Chapter Notes

Some materials from this chapter are derived from the materials courtesy of **Dr Cheng Holun, Alan** from School of Computing, National University of Singapore.

There are many other Convex Hull algorithms such as **Jarvis's March**, **Gift Wrapping**, and **Upper/Lower Hull** with more or less similar time complexity as Graham's Scan. The Graham's Scan algorithm presented in this chapter is usually enough for most contest problems.

There is a computational geometry technique that has not been discussed yet: **plane sweep**. Interested reader should consult the following books [19, 6, 4].

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. He must train himself to be familiar with many degenerate (special) cases in certain geometry problems, able to deal with precision errors, etc.

There are approximately **96 programming exercises** discussed in this chapter.

Appendix A

Problem Credits

The problems discussed in this book are mainly taken from UVa online judge [17] and ACM ICPC Live Archive [11]. We have tried our best to contact the original authors and get their permissions.

So far, we have contacted the following problem setters and obtained their permissions: Sohel Hafiz, Shahriar Manzoor, Manzurur Rahman Khan, Rujia Liu, Gordon Cormack, Jim Knisely, Melvin Zhang, and Colin Tan.

If any of the author of a particular problem discussed in this book that we have not contacted yet does not allow his/her problem to be used in this book, we will replace that particular problem with another similar problem from different problem setter.

Appendix B

We Want Your Feedbacks

You, the reader, can help us to improve the quality of the future versions 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 future version of this book (i.e. I have a better example/algorithm to illustrate a certain point), etc, please send email the main author directly: stevenhalim@gmail.com.

Bibliography

[1] Ahmed Shamsul Arefin. *Art of Programming Contest (from Steven's Website)*. Gyankosh Prokashoni (Available Online), 2006.

[2] Jon Bentley. *Programming Pearls*. Addison Wesley, 2nd edition, 2000.

[3] Frank Carrano. *Data Abstraction & Problem Solving with C++*. Pearson, 5th edition, 2007.

[4] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Cliff Stein. *Introduction to Algorithm*. MIT Press, 2nd edition, 2001.

[5] Sanjoy Dasgupta, Christos Papadimitriou, and Umesh Vazirani. *Algorithms*. McGraw Hill, 2008.

[6] Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Cheong Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer, 2nd edition, 2000.

[7] Project Euler. Project Euler.
http://projecteuler.net/.

[8] Michal Forišek. IOI Syllabus.
http://people.ksp.sk/ misof/ioi-syllabus/ioi-syllabus-2009.pdf.

[9] 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.

[10] Steven Halim, Roland Hock Chuan Yap, and Felix Halim. Engineering Stochastic Local Search for the Low Autocorrelation Binary Sequence Problem. In *Principles and Practice of Constraint Programming*, pages 640–645, 2008.

[11] Competitive Learning Institute. ACM ICPC Live Archive.
http://acm.uva.es/archive/nuevoportal.

[12] IOI. International Olympiad in Informatics.
http://ioinformatics.org.

[13] Jon Kleinberg and Eva Tardos. *Algorithm Design*. Addison Wesley, 2006.

[14] Anany Levitin. *Introduction to The Design & Analysis of Algorithms*. Addison Wesley, 1st edition, 2002.

[15] Rujia Liu. *Algorithm Contests for Beginners (In Chinese)*. Tsinghua University Press, 2009.

[16] Rujia Liu and Liang Huang. *The Art of Algorithms and Programming Contests (In Chinese).* Tsinghua University Press, 2003.

[17] University of Valladolid. Online Judge.
http://uva.onlinejudge.org.

[18] USA Computing Olympiad. USACO Training Program Gateway.
http://train.usaco.org/usacogate.

[19] Joseph O'Rourke. *Computational Geometry in C.* Cambridge University Press, 2nd edition, 1998.

[20] Kenneth H. Rosen. *Elementary Number Theory and its applications.* Addison Wesley Longman, 4th edition, 2000.

[21] Robert Sedgewick. *Algorithms in C++, Part 1-5.* Addison Wesley, 3rd edition, 2002.

[22] Steven S Skiena. *The Algorithm Design Manual.* Springer, 2008.

[23] Steven S. Skiena and Miguel A. Revilla. *Programming Challenges.* Springer, 2003.

[24] Wing-Kin Sung. *Algorithms in Bioinformatics: A Practical Introduction.* CRC Press (Taylor & Francis Group), 1st edition, 2010.

[25] TopCoder. Algorithm Tutorials.
http://www.topcoder.com/tc?d1=tutorials&d2=alg_index&module=Static.

[26] TopCoder. Single Round Match (SRM).
http://www.topcoder.com/tc.

[27] Baylor University. ACM International Collegiate Programming Contest.
http://icpc.baylor.edu/icpc.

[28] Tom Verhoeff. 20 Years of IOI Competition Tasks. *Olympiads in Informatics*, 3:149166, 2009.

[29] Adrian Vladu and Cosmin Negruşeri. Suffix arrays - a programming contest approach. 2008.

[30] Wikipedia. Catalan number.
http://en.wikipedia.org/wiki/Catalan_number.

[31] Wikipedia. Combinatorics.
http://en.wikipedia.org/wiki/Combinatorics.

[32] Wikipedia. Cycle-Finding (Detection).
http://en.wikipedia.org/wiki/Cycle_detection.

[33] Wikipedia. Disjoint-set data structure.
http://en.wikipedia.org/wiki/Disjoint-set_data_structure.

[34] Wikipedia. Edmond's matching algorithm.
http://en.wikipedia.org/wiki/Edmonds's_matching_algorithm.

[35] Wikipedia. Eight queens puzzle.
http://en.wikipedia.org/wiki/Eight_queens_puzzle.

[36] Wikipedia. Euler's totient function.
http://en.wikipedia.org/wiki/Euler's_totient_function.

[37] Wikipedia. Gaussian Elimination for Solving System of Linear Equations.
http://en.wikipedia.org/wiki/Gaussian_elimination.

[38] Wikipedia. Great-Circle Distance.
http://en.wikipedia.org/wiki/Great_circle_distance.

[39] Wikipedia. Longest Path Problem.
http://en.wikipedia.org/wiki/Longest_path_problem.

[40] Wikipedia. Lowest Common Ancestor.
http://en.wikipedia.org/wiki/Lowest_common_ancestor.

[41] Wikipedia. Miller-Rabin Primality Test.
http://en.wikipedia.org/wiki/Miller-Rabin_primality_test.

[42] Wikipedia. Path Cover.
http://en.wikipedia.org/wiki/Path_cover.

[43] Wikipedia. Pick's Theorem.
http://en.wikipedia.org/wiki/Pick's_theorem.

[44] Wikipedia. Sieve of Eratosthenes.

[45] Wikipedia. Tarjan's Strongly Connected Components Algorithm.
http://en.wikipedia.org/wiki/Tarjan's_strongly_connected_components_algorithm.

[46] Wikipedia. The Free Encyclopedia.
http://en.wikipedia.org.

[47] Yonghui Wu and Jiang De Wang. *Practical Algorithm Analysis and Program Design (In Chinese)*. Posts and Telecom Press, 2009.

Index