- fellow Teaching Assistant of CS3233 @ NUS in the past two years: Harta Wijaya, Trinh Tuan Phuong, and Huang Da.
- my CS3233 students in Sem2 AY2011/2012 who contributed in both technical and presentation aspects of the second edition of this book, in alphabetical order: Cao Sheng, Chua Wei Kuan, Han Yu, Huang Da, Huynh Ngoc Tai, Ivan Reinaldo, John Goh Choo Ern, Le Viet Tien, Lim Zhi Qin, Nalin Ilango, Nguyen Hoang Duy, Nguyen Phi Long, Nguyen Quoc Phong, Pallav Shinghal, Pan Zhengyang, Pang Yan Han, Song Yangyu, Tan Cheng Yong Desmond, Tay Wenbin, Yang Mansheng, Zhao Yang, Zhou Yiming, and two other students who prefer to be anonymous.
- the proof readers: Six of CS3233 students in Sem2 AY2011/2012 (underlined) and Hubert Teo Hua Kian.
- my CS3233 students in Sem2 AY2012/2013 who contributed in both technical and presentation aspects of the second edition of this book, in alphabetical order: Arnold Christopher Koroa, Cao Luu Quang, Lim Puay Ling Pauline, Erik Alexander Qvick Faxaa, Jonathan Darryl Widjaja, Nguyen Tan Sy Nguyen, Nguyen Truong Duy, Ong Ming Hui, Pan Yuxuan, Shubham Goyal, Sudhanshu Khemka, Tang Binbin, Trinh Ngoc Khanh, Yao Yujian, Zhao Yue, and Zheng Naijia.

- the NUS Centre for Development of Teaching and Learning (CDTL) for giving the initial funding to build the algorithm visualization website.
- my wife Grace Suryani and my daughter Jane Angelina for your love in our family.

To a better future of humankind,
Steven and Felix Halim
Singapore, 24 May 2013

Copyright

Authors' Profiles

Steven Halim, PhD[1]

stevenhalim@gmail.com

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

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

Felix Halim, PhD[2]

felix.halim@gmail.com

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

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

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

Abbreviations

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

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

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

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

ED : Edit Distance

FIFO : First In First Out
FT : Fenwick Tree

GCD : Greatest Common Divisor

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

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

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

OJ : Online Judge

PE : Presentation Error

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

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

TLE : Time Limit Exceeded

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

WA : Wrong Answer
WF : World Finals

List of Tables

List of Figures

Chapter 1

Introduction

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

1.1 Competitive Programming

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

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

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

Abridged Problem Description:

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

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

Sample output:
$cost = 4.83$.

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

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

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

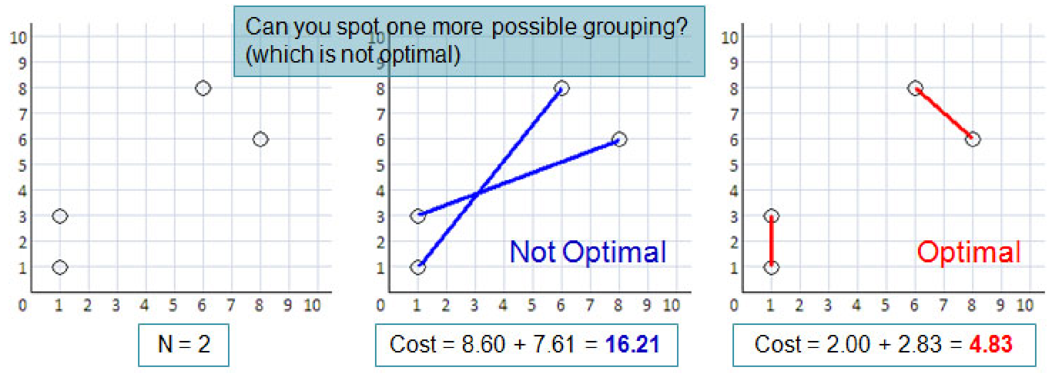


Figure 1.1: Illustration of UVa 10911 - Forming Quiz Teams

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

- Uncompetitive programmer A (a.k.a. the blurry one):
  Step 1: Reads the problem and becomes confused. (This problem is new for him).
  Step 2: Tries to code something: Reading the non-trivial input and output.
  Step 3: Realizes that all his attempts are *not* **Accepted (AC)**:
  Greedy (Section 3.4): Repeatedly pairing the two remaining students with the shortest separating distances gives the **Wrong Answer (WA)**.
  Naïve **Complete Search**: Using recursive backtracking (Section 3.2) and trying all possible pairings yields **Time Limit Exceeded (TLE)**.
- Uncompetitive programmer B (Give up):
  Step 1: Reads the problem and realizes that he has seen this problem before.
  But also remembers that he has not learned how to solve this kind of problem...
  He is not aware of the **Dynamic Programming (DP)** solution (Section 3.5)...
  Step 2: Skips the problem and reads another problem in the problem set.
- (Still) Uncompetitive programmer C (Slow):
  Step 1: Reads the problem and realizes that it is a hard problem: **'minimum weight perfect matching on a small general weighted graph'**. However, since the input size is small, this problem is solvable using DP. The DP state is a **bitmask** that describes a matching status, and matching unmatched students i and j will turn on two bits i and j in the bitmask (Section 8.3.1).
  Step 2: Codes I/O routine, writes recursive top-down DP, tests, **debugs** >.<...
  Step 3: *After 3 hours*, his solution obtains AC (passed all the secret test data).
- Competitive programmer D:
  Completes all the steps taken by uncompetitive programmer C in ≤ 30 minutes.
- Very competitive programmer E:
  A very competitive programmer (e.g. the red 'target' coders in TopCoder [32]) would solve this 'well known' problem ≤ 15 minutes...

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

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

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

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

1.2 Tips to be Competitive

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

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

1.2.1 Tip 1: Type Code Faster!

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

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

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

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

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

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

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

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

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

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

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

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

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

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

1.2.2 Tip 2: Quickly Identify Problem Types

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

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

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

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

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

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

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

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

Table 1.2: Problem Types (Compact Form)

To be *competitive*, that is, *do well* in a programming contest, you must be able to confidently and frequently classify problems as type A and minimize the number of problems that you classify into type B. That is, you need to acquire sufficient algorithm knowledge and develop your programming skills so that you consider many classical problems to be easy. However, to *win* a programming contest, you will also need to develop sharp *problem solving skills* (e.g. reducing the given problem to a known problem, identifying subtle hints or special properties in the problem, attacking the problem from a non obvious angle, etc) so that you (or your team) will be able to derive the required solution to a hard/original type C problem in IOI or ICPC Regionals/World Finals and do so *within* the duration of the contest.

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

Table 1.3: Exercise: Classify These UVa Problems

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

1.2.3 Tip 3: Do Algorithm Analysis

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

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

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

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

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

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

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

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

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

- Basic time and space complexity analysis for iterative and recursive algorithms:
  - An algorithm with k-nested loops of about n iterations each has $O(n^k)$ complexity.
  - If your algorithm is recursive with b recursive calls per level and has L levels, the algorithm has roughly $O(b^L)$ complexity, but this is a only a rough upper bound. The actual complexity depends on what actions are done per level and whether pruning is possible.
  - A Dynamic Programming algorithm or other iterative routine which processes a 2D $n \times n$ matrix in $O(k)$ per cell runs in $O(k \times n^2)$ time. This is explained in further detail in Section 3.5.
- More advanced analysis techniques:
  - Prove the correctness of an algorithm (especially for Greedy algorithms in Section 3.4), to minimize your chance of getting the 'Wrong Answer' verdict.
  - Perform the amortized analysis (e.g. see Chapter 17 of [7])—although rarely used in contests—to minimize your chance of getting the 'Time Limit Exceeded' verdict, or worse, considering your algorithm to be too slow and skips the problem when it is in fact fast enough in amortized sense.
  - Do output-sensitive analysis to analyze algorithm which (also) depends on output size and minimize your chance of getting the 'Time Limit Exceeded' verdict. For example, an algorithm to search for a string with length m in a long string with the help of a Suffix Tree (that is already built) runs in $O(m+occ)$ time. The time taken for this algorithm to run depends not only on the input size m but also the output size—the number of occurrences occ (see more details in Section 6.6).

- Familiarity with these bounds:
  - $2^{10} = 1,024 \approx 10^3$, $2^{20} = 1,048,576 \approx 10^6$.
  - 32-bit signed integers (`int`) and 64-bit signed integers (`long long`) have upper limits of $2^{31}-1 \approx 2\times10^9$ (safe for up to $\approx$ 9 decimal digits) and $2^{63}-1 \approx 9\times10^{18}$ (safe for up to $\approx$ 18 decimal digits) respectively.
  - Unsigned integers can be used if only non-negative numbers are required. 32-bit unsigned integers (`unsigned int`) and 64-bit unsigned integers (`unsigned long long`) have upper limits of $2^{32}-1 \approx 4\times10^9$ and $2^{64}-1 \approx 1.8\times10^{19}$ respectively.
  - If you need to store integers $\geq 2^{64}$, use the Big Integer technique (Section 5.3).
  - There are $n!$ permutations and 2^n subsets (or combinations) of n elements.
  - The best time complexity of a comparison-based sorting algorithm is $\Omega(n \log_2 n)$.
  - Usually, $O(n \log_2 n)$ algorithms are sufficient to solve most contest problems.
  - The largest input size for typical programming contest problems must be $< 1M$. Beyond that, the time needed to read the input (the Input/Output routine) will be the bottleneck.
  - A typical year 2013 CPU can process $100M = 10^8$ operations in a few seconds.

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

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

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

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

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

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

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

1. There are n webpages ($1 \leq n \leq 10M$). Each webpage i has a page rank r_i. You want to pick the top 10 pages with the highest page ranks. Which method is better?
   (a) Load all n webpages' page rank to memory, sort (Section 2.2) them in descending page rank order, obtaining the top 10.
   (b) Use a priority queue data structure (a heap) (Section 2.3).
2. Given an $M \times N$ integer matrix Q ($1 \leq M, N \leq 30$), determine if there exists a sub-matrix of Q of size $A \times B$ ($1 \leq A \leq M, 1 \leq B \leq N$) where `mean`$(Q) = 7$.
   (a) Try all possible sub-matrices and check if the mean of each sub-matrix is 7. This algorithm runs in $O(M^3 \times N^3)$.
   (b) Try all possible sub-matrices, but in $O(M^2 \times N^2)$ with this technique: ________ .
3. Given a list L with $10K$ integers, you need to *frequently* obtain `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).
   (b) Simple Array pre-processed with Dynamic Programming (Section 2.2 & 3.5).
   (c) Balanced Binary Search Tree (Section 2.3).
   (d) Binary Heap (Section 2.3).
   (e) Segment Tree (Section 2.4.3).
   (f) Binary Indexed (Fenwick) Tree (Section 2.4.4).
   (g) Suffix Tree (Section 6.6.2) or its alternative, Suffix Array (Section 6.6.4).
4. Given a set S of N points *randomly* scattered on a 2D plane ($2 \leq N \leq 1000$), find two points $\in S$ that have the greatest separating Euclidean distance. Is an $O(N^2)$ complete search algorithm that tries all possible pairs feasible?
   (a) Yes, such complete search is possible.
   (b) No, we must find another way. We must use: ________ .
5. You have to compute the shortest path between two vertices on a weighted Directed Acyclic Graph (DAG) with $|V|, |E| \leq 100K$. Which algorithm(s) can be used in a typical programming contest (that is, with a time limit of approximately 3 seconds)?
   (a) Dynamic Programming (Section 3.5, 4.2.5, & 4.7.1).
   (b) Breadth First Search (Section 4.2.2 & 4.4.2).
   (c) Dijkstra's (Section 4.4.3).

   (d) Bellman Ford's (Section 4.4.4).
   (e) Floyd Warshall's (Section 4.5).

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

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

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

   (a) Test if $n!$ % $m == 0$.
   (b) The naïve approach above will not work, use: _________ (Section 5.5.1).

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

   (a) The complete search mentioned in question 3 can still be used.
   (b) The naïve approach above will not work, use: _________ (Section 7.3.7).

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

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

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

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

1.2.4 Tip 4: Master Programming Languages

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

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

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

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

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

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

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

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

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

One possible solution is as follows:

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

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

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

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

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

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

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

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

1. Using **Java**, read in a double
   (e.g. `1.4732`, `15.324547327`, etc.)
   echo it, but with a minimum field width of 7 and 3 digits after the decimal point
   (e.g. `ss1.473` (where '`s`' denotes a space), `s15.325`, etc.)
2. Given an integer n ($n \leq 15$), print π to n digits after the decimal point (rounded).
   (e.g. for $n = 2$, print `3.14`; for $n = 4$, print `3.1416`; for $n = 5$, prints `3.14159`.)
3. Given a date, determine the day of the week (Monday, ..., Sunday) on that day.
   (e.g. 9 August 2010—the launch date of the first edition of this book—is a Monday.)
4. Given n random integers, print the distinct (unique) integers in sorted order.
5. Given the distinct and valid birthdates of n people as triples (DD, MM, YYYY), order them first by ascending birth months (MM), then by ascending birth dates (DD), and finally by ascending age.
6. Given a list of *sorted* integers L of size up to $1M$ items, determine whether a value v exists in L with no more than 20 comparisons (more details in Section 2.2).
7. Generate all possible permutations of {'A', 'B', 'C', ..., 'J'}, the first $N = 10$ letters in the alphabet (see Section 3.2.1).
8. Generate all possible subsets of {0, 1, 2, ..., N-1}, for $N = 20$ (see Section 3.2.1).
9. Given a string that represents a base X number, convert it to an equivalent string in base Y, $2 \leq X, Y \leq 36$. For example: "FF" in base X = 16 (hexadecimal) is "255" in base $Y_1 = 10$ (decimal) and "11111111" in base $Y_2 = 2$ (binary). See Section 5.3.2.
10. Let's define a 'special word' as a lowercase alphabet followed by two consecutive digits.
    Given a string, replace all 'special words' of length 3 with 3 stars "***", e.g.
    S = "line: a70 and z72 will be replaced, aa24 and a872 will not"
    should be transformed into:
    S = "line: *** and *** will be replaced, aa24 and a872 will not".
11. Given a *valid* mathematical expression involving '+', '-', '*', '/', '(', and ')' in a single line, evaluate that expression. (e.g. a rather complicated but valid expression `3 + (8 - 7.5) * 10 / 5 - (2 + 5 * 7)` should produce `-33.0` when evaluated with standard operator precedence.)

1.2.5 Tip 5: Master the Art of Testing Code

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

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

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

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

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

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

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

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

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

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

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

4. Your test cases should include *large* cases. Increase the input size incrementally up to the maximum input bounds stated in the problem description. Use large test cases with trivial structures that are easy to verify with manual computation and large *random* test cases to test if your code terminates in time and still produces reasonable output (since the correctness would be difficult to verify here). Sometimes your program may work for small test cases, but produces wrong answer, crashes, or exceeds the time limit when the input size increases. If that happens, check for overflows, out of bound errors, or improve your algorithm.
5. Though this is rare in modern programming contests, do not assume that the input will always be nicely formatted if the problem description does not explicitly state it (especially for a badly written problem). Try inserting additional whitespace (spaces, tabs) in the input and test if your code is still able to obtain the values correctly without crashing.

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

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

1. You receive a WA verdict for a very easy problem. What should you do?
   (a) Abandon this problem for another.
   (b) Improve the performance of your solution (code optimizations/better algorithm).
   (c) Create tricky test cases to find the bug.
   (d) (In team contests): Ask your team mate to re-do the problem.
2. You receive a TLE verdict for your $O(N^3)$ solution.
   However, the maximum N is just 100. What should you do?
   (a) Abandon this problem for another.
   (b) Improve the performance of your solution (code optimizations/better algorithm).
   (c) Create tricky test cases to find the bug.
3. Follow up to Question 2: What if the maximum N is 100.000?
4. Another follow up to Question 2: What if the maximum N is 1.000, the output only depends on the size of input N, and you still have *four hours* of competition time left?
5. You receive an RTE verdict. Your code (seems to) execute perfectly on your machine. What should you do?

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

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

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

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

   (a) Abandon the problem with the WA code, switch to the other problem in an attempt to solve one more problem.
   (b) Insist that you have to debug the WA code. There is not enough time to start working on a new problem.
   (c) (In ICPC): Print the WA code. Ask two other team members to scrutinize it while you switch to that other problem in an attempt to solve *two* more problems.

1.2.6 Tip 6: Practice and More Practice

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

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

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

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

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

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

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

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

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

1.2.7 Tip 7: Team Work (for ICPC)

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

- Practice coding on blank paper. (This is useful when your teammate is using the computer. When it is your turn to use the computer, you can then just type the code as fast as possible rather than spending time thinking in front of the computer.)
- The 'submit and print' strategy: If your code gets an AC verdict, ignore the printout. If it still is not AC, debug your code using that printout (and let your teammate uses the computer for other problem). Beware: Debugging without the computer is not an easy skill to master.
- If your teammate is currently coding his algorithm, prepare challenges for his code by preparing hard corner-case test data (hopefully his code passes all those).
- The X-factor: Befriend your teammates *outside* of training sessions and contests.

1.3 Getting Started: The Easy Problems

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

1.3.1 Anatomy of a Programming Contest Problem

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

- **Background story/problem description**. Usually, the easier problems are written to *deceive* contestants and made to appear difficult, for example by adding 'extra information' to create a diversion. Contestants should be able to *filter out* these unimportant details and focus on the essential ones. For example, the entire opening paragraphs except the last sentence in UVa 579 - ClockHands are about the history of the clock and is completely unrelated to the actual problem. However, harder problems are usually written as succinctly as possible—they are already difficult enough without additional embellishment.

- **Input and Output description**. In this section, you will be given details on how the input is formatted and on how you should format your output. This part is usually written in a formal manner. A good problem should have clear input constraints as the same problem might be solvable with different algorithms for different input constraints (see Table 1.4).
- **Sample Input and Sample Output**. Problem authors usually only provide trivial test cases to contestants. The sample input/output is intended for contestants to check their basic understanding of the problem and to verify if their code can parse the given input using the given input format and produce the correct output using the given output format. Do not submit your code to the judge if it does not even pass the given sample input/output. See Section 1.2.5 about testing your code before submission.
- **Hints or Footnotes**. In some cases, the problem authors may drop hints or add footnotes to further describe the problem.

1.3.2 Typical Input/Output Routines

Multiple Test Cases

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

- The number of test cases is given in the first line of the input.
- The multiple test cases are terminated by special values (usually zeroes).
- The multiple test cases are terminated by the EOF (end-of-file) signal.

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

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

Case Numbers and Blank Lines

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

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

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

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

Variable Number of Inputs

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

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

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

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

1.3.3 Time to Start the Journey

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

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

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

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

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

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

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

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

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

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

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

1.4 The Ad Hoc Problems

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

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

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

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

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

The categories:

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

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

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

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

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

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

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

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

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

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

- Ad Hoc problems in **other chapters**
  There are many other Ad Hoc problems which we have shifted to other chapters since they required knowledge above basic programming skills.
  - Ad Hoc problems involving the usage of basic linear data structures (especially arrays) are listed in Section 2.2.
  - Ad Hoc problems involving mathematical computation are listed in Section 5.2.
  - Ad Hoc problems involving string processing are listed in Section 6.3.
  - Ad Hoc problems involving basic geometry are listed in Section 7.2.
  - Ad Hoc problems listed in Chapter 9.

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

Programming Exercises related to Ad Hoc problems:

- Game (Card)
  1. UVa 00162 - Beggar My Neighbour (card game simulation; straightforward)
  2. **UVa 00462 - Bridge Hand Evaluator *** (simulation; card)
  3. UVa 00555 - Bridge Hands (card game)
  4. UVa 10205 - Stack 'em Up (card game)
  5. *UVa 10315 - Poker Hands* (tedious problem)
  6. **UVa 10646 - What is the Card? *** (shuffle cards with some rule and then get certain card)
  7. UVa 11225 - Tarot scores (another card game)
  8. UVa 11678 - Card's Exchange (actually just an array manipulation problem)
  9. ***UVa 12247 - Jollo **** (interesting card game; simple, but requires good logic to get all test cases correct)
- Game (Chess)
  1. UVa 00255 - Correct Move (check the validity of chess moves)
  2. **UVa 00278 - Chess *** (ad hoc, chess, closed form formula exists)
  3. **UVa 00696 - How Many Knights *** (ad hoc, chess)
  4. UVa 10196 - Check The Check (ad hoc chess game, tedious)
  5. **UVa 10284 - Chessboard in FEN *** (FEN = Forsyth-Edwards Notation is a standard notation for describing board positions in a chess game)
  6. UVa 10849 - Move the bishop (chess)
  7. UVa 11494 - Queen (ad hoc, chess)
- Game (Others), Easier
  1. UVa 00340 - Master-Mind Hints (determine strong and weak matches)
  2. **UVa 00489 - Hangman Judge *** (just do as asked)
  3. *UVa 00947 - Master Mind Helper* (similar to UVa 340)
  4. **UVa 10189 - Minesweeper *** (simulate Minesweeper, similar to UVa 10279)
  5. UVa 10279 - Mine Sweeper (a 2D array helps, similar to UVa 10189)
  6. UVa 10409 - Die Game (just simulate the die movement)
  7. UVa 10530 - Guessing Game (use a 1D flag array)
  8. **UVa 11459 - Snakes and Ladders *** (simulate it, similar to UVa 647)
  9. *UVa 12239 - Bingo* (try all 90^2 pairs, see if all numbers in `[0..N]` are there)

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

- Interesting Real Life Problems, Harder (more tedious)
  1. UVa 00139 - Telephone Tangles (calculate phone bill; string manipulation)
  2. UVa 00145 - Gondwanaland Telecom (similar nature with UVa 139)
  3. *UVa 00333 - Recognizing Good ISBNs* (note: this problem has 'buggy' test data with blank lines that potentially cause lots of 'Presentation Errors')
  4. UVa 00346 - Getting Chorded (musical chord, major/minor)
  5. **UVa 00403 - Postscript** * (emulation of printer driver, tedious)
  6. *UVa 00447 - Population Explosion* (life simulation model)
  7. UVa 00448 - OOPS (tedious 'hexadecimal' to 'assembly language' conversion)
  8. *UVa 00449 - Majoring in Scales* (easier if you have a musical background)
  9. UVa 00457 - Linear Cellular Automata (simplified 'game of life' simulation; similar idea with UVa 447; explore the Internet for that term)
  10. UVa 00538 - Balancing Bank Accounts (the problem's premise is quite true)
  11. **UVa 00608 - Counterfeit Dollar** * (classical problem)
  12. UVa 00706 - LC-Display (what we see in old digital display)
  13. ***UVa 01061 - Consanguine Calculations*** * (LA 3736 - WorldFinals Tokyo07, consanguine = blood; this problem asks possible combinations of blood types and Rh factor; solvable by trying all eight possible blood + Rh types with the information given in the problem description)
  14. UVa 10415 - Eb Alto Saxophone Player (about musical instruments)
  15. UVa 10659 - Fitting Text into Slides (typical presentation programs do this)
  16. UVa 11223 - O: dah, dah, dah (tedious morse code conversion)
  17. UVa 11743 - Credit Check (Luhn's algorithm to check credit card numbers; search the Internet to learn more)
  18. *UVa 12342 - Tax Calculator* (tax computation can be tricky indeed)
- Time
  1. UVa 00170 - Clock Patience (simulation, time)
  2. UVa 00300 - Maya Calendar (ad hoc, time)
  3. **UVa 00579 - Clock Hands** * (ad hoc, time)
  4. **UVa 00893 - Y3K** * (use Java GregorianCalendar; similar to UVa 11356)
  5. UVa 10070 - Leap Year or Not Leap ... (more than ordinary leap years)
  6. *UVa 10339 - Watching Watches* (find the formula)
  7. UVa 10371 - Time Zones (follow the problem description)
  8. UVa 10683 - The decadary watch (simple clock system conversion)
  9. UVa 11219 - How old are you? (be careful with boundary cases!)
  10. UVa 11356 - Dates (very easy if you use Java GregorianCalendar)
  11. UVa 11650 - Mirror Clock (some mathematics required)
  12. UVa 11677 - Alarm Clock (similar idea with UVa 11650)
  13. **UVa 11947 - Cancer or Scorpio** * (easier with Java GregorianCalendar)
  14. UVa 11958 - Coming Home (be careful with 'past midnight')
  15. UVa 12019 - Doom's Day Algorithm (Gregorian Calendar; get DAY_OF_WEEK)
  16. UVa 12136 - Schedule of a Married Man (LA 4202, Dhaka08, check time)
  17. *UVa 12148 - Electricity* (easy with Gregorian Calendar; use method 'add' to add one day to previous date and see if it is the same as the current date)
  18. *UVa 12439 - February 29* (inclusion-exclusion; lots of corner cases; be careful)
  19. *UVa 12531 - Hours and Minutes* (angles between two clock hands)

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

1.5 Solutions to Non-Starred Exercises

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

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

Exercise 1.2.1: The complete Table 1.3 is shown below.

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

Exercise 1.2.2: The answers are:

1. (b) Use a priority queue data structure (heap) (Section 2.3).
2. (b) Use 2D Range Sum Query (Section 3.5.2).
3. If list L is static, (b) Simple Array that is pre-processed with Dynamic Programming (Section 2.2 & 3.5). If list L is dynamic, then (g) Fenwick Tree is a better answer (easier to implement than (f) Segment Tree).
4. (a) Yes, a complete search is possible (Section 3.2).
5. (a) $O(V + E)$ Dynamic Programming (Section 3.5, 4.2.5, & 4.7.1).
   However, (c) $O((V + E) \log V)$ Dijkstra's algorithm is also possible since the extra $O(\log V)$ factor is still 'small' for V up to $100K$.
6. (a) Sieve of Eratosthenes (Section 5.5.1).
7. (b) The naïve approach above will not work. We must (prime) factorize $n!$ and m and see if the (prime) factors of m can be found in the factors of $n!$ (Section 5.5.5).
8. (b) No, we must find another way. First, find the Convex Hull of the N points in $O(n \log n)$ (Section 7.3.7). Let the number of points in $CH(S) = k$. As the points are randomly scattered, k will be much smaller than N. Then, find the two farthest points by examining all pairs of points in the $CH(S)$ in $O(k^2)$.
9. (b) The naïve approach is too slow. Use KMP or Suffix Array (Section 6.4 or 6.6)!

Exercise 1.2.3: The Java code is shown below:

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

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

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

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

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

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

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

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

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

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

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

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

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

Exercise 1.2.4: Situational considerations are in brackets:

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

2. You receive a TLE verdict for your $O(N^3)$ solution.
   However, the maximum N is just 100. What should you do?
   (a) Abandon this problem for another. **(Not ok, your team will lose out.)**
   (b) Improve the performance of your solution. **(Not ok, we should not get TLE with an $O(N^3)$ algorithm if $N \leq 400$.)**
   (c) Create tricky test cases to find the bug. **(This is the answer—maybe your program runs into an accidental infinite loop in some test cases.)**

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

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

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

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

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

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

9. There is one hour to go before the end of the contest. You have 1 WA code and 1 fresh idea for *another* problem. What should you (or your team) do?
(In chess terminology, this is called the 'end game' situation.)
   (a) Abandon the problem with the WA code, switch to the other problem in an attempt to solve one more problem.**(OK in individual contests like IOI.)**
   (b) Insist that you have to debug the WA code. There is not enough time to start working on a new problem. **(If the idea for another problem involves complex and tedious code, then deciding to focus on the WA code may be a good idea rather than having two incomplete/'non AC' solutions.)**
   (c) (In ICPC): Print the WA code. Ask two other team members to scrutinize it while you switch to that other problem in an attempt to solve *two* more problems.
   (If the solution for the other problem can be coded in less than 30 minutes, then implement it while your team mates try to find the bug for the WA code by studying the printed copy.)

Data Abstraction & Problem Solving with C++
Algorithms in C++
The Design & Analysis of Algorithms
Introduction to Algorithms
Algorithm Design
Algorithms
The Algorithm Design Manual
Programming Challenges
Art of Programming Contest
Olympiads in Informatics
Hacker's Delight
Elementary Number Theory
Algorithms in Bioinformatics
Computational Geometry in C
Computational Geometry

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

1.6 Chapter Notes

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

- To improve your typing skill as mentioned in Tip 1, you may want to play the many typing games available online.
- Tip 2 is adapted from the introduction text in USACO training gateway [48].
- More details about Tip 3 can be found in many CS books, e.g. Chapter 1-5, 17 of [7].
- Online references for Tip 4:
  `http://www.cppreference.com` and `http://www.sgi.com/tech/stl/` for C++ STL;
  `http://docs.oracle.com/javase/7/docs/api/` for Java API.
  You do not have to memorize all library functions, but it is useful to memorize functions that you frequently use.
- For more insights on better testing (Tip 5), a slight detour to software engineering books may be worth trying.
- There are many other Online Judges apart from those mentioned in Tip 6, e.g.
  - Codeforces, `http://codeforces.com/`,
  - Peking University Online Judge, (POJ) `http://poj.org`,
  - Zhejiang University Online Judge, (ZOJ) `http://acm.zju.edu.cn`,
  - Tianjin University Online Judge, `http://acm.tju.edu.cn/toj`,
  - Ural State University (Timus) Online Judge, `http://acm.timus.ru`,
  - URI Online Judge, `http://www.urionlinejudge.edu.br`, etc.
- For a note regarding team contest (Tip 7), read [16].

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

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

Chapter 2

Data Structures and Libraries

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

2.1 Overview and Motivation

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

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

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

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

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

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

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

Value-Added Features of this Book

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

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

Source code: `chx_yy_name.cpp/java`

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

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

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

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

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

2.2 Linear DS with Built-in Libraries

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

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

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

  Source code: `ch2_01_array_vector.cpp/java`

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

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

  1. $O(n^2)$ comparison-based sorting algorithms: Bubble/Selection/Insertion Sort, etc. These algorithms are (awfully) slow and usually avoided in programming contests, though understanding them might help you solve certain problems.
  2. $O(n \log n)$ comparison-based sorting algorithms: Merge/Heap/Quick Sort, etc. These algorithms are the default choice in programming contests as an $O(n \log n)$ complexity is optimal for comparison-based sorting. Therefore, these sorting algorithms run in the 'best possible' time in most cases (see below for special purpose sorting algorithms). In addition, these algorithms are well-known and hence we do not need to 'reinvent the wheel'[5]—we can simply use `sort`, `partial_sort`, or `stable_sort` in C++ STL `algorithm` (or `Collections.sort` in Java) for standard sorting tasks. We only need to specify the required comparison function and these library routines will handle the rest.
  3. Special purpose sorting algorithms: $O(n)$ Counting/Radix/Bucket Sort, etc. Although rarely used, these special purpose algorithms are good to know as they can reduce the required sorting time if the data has certain special characteristics. For example, Counting Sort can be applied to integer data that lies in a small range (see Section 9.32).

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

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

1. $O(n)$ Linear Search: Consider every element from index 0 to index $n-1$ (avoid this whenever possible).
2. $O(\log n)$ Binary Search: Use `lower_bound`, `upper_bound`, or `binary_search` in C++ STL `algorithm` (or Java `Collections.binarySearch`). If the input array is unsorted, it is necessary to sort the array at least once (using one of the $O(n \log n)$ sorting algorithm above) before executing one (or *many*) Binary Search(es).
3. $O(1)$ with Hashing: This is a useful technique to use when fast access to known values are required. If a suitable hash function is selected, the probability of a collision to be made is negligibly small. Still, this technique is rarely used and we can live without it[6] for most (contest) problems.

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

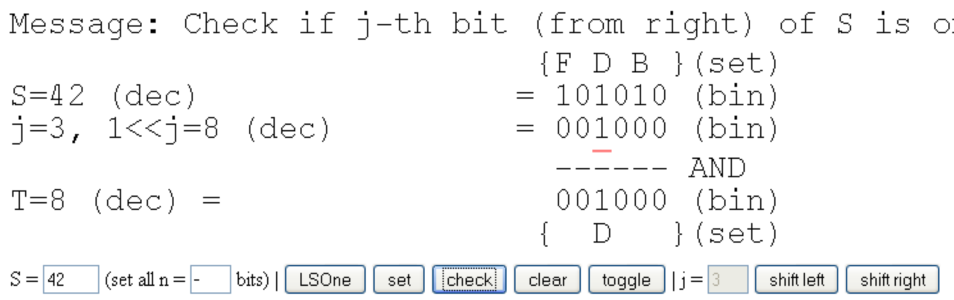
Source code: `ch2_02_algorithm_collections.cpp/java`

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

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

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

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



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

S = 42 (set all n = - bits) | j = 3

Figure 2.1: Bitmask Visualization

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

[6]However, questions about hashing frequently appear in interviews for IT jobs.

[7]To avoid issues with the two's complement representation, use a 32-bit/64-bit *signed* integer to represent bitmasks of up to 30/62 items only, respectively.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Source code: `ch2_03_bit_manipulation.cpp/java`

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

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

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

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

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

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

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

Source code: `ch2_04_stack_queue.cpp/java`

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

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

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

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

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

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

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

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

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

1. Obtain the remainder (modulo) of S when it is divided by N (N is a power of 2) e.g. $S = (7)_{10} \ \% \ (4)_{10} = (111)_2 \ \% \ (100)_2 = (11)_2 = (3)_{10}$.
2. Determine if S is a power of 2. e.g. $S = (7)_{10} = (111)_2$ is not a power of 2, but $(8)_{10} = (100)_2$ is a power of 2.
3. Turn off the last bit in S, e.g. $S = (40)_{10} = (10\underline{1}000)_2 \rightarrow S = (32)_{10} = (10\underline{0}000)_2$.
4. Turn on the last zero in S, e.g. $S = (41)_{10} = (1010\underline{0}1)_2 \rightarrow S = (43)_{10} = (1010\underline{1}1)_2$.
5. Turn off the last consecutive run of ones in S e.g. $S = (39)_{10} = (100\underline{111})_2 \rightarrow S = (32)_{10} = (100\underline{000})_2$.
6. Turn on the last consecutive run of zeroes in S e.g. $S = (36)_{10} = (1001\underline{00})_2 \rightarrow S = (39)_{10} = (1001\underline{11})_2$.

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

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

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

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

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

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