Contents

Preface

This book is a must have for every competitive programmer to master during their middle phase of their programming career if they wish to take a leap forward from being just another ordinary coder to being among one of the top finest programmer in the world.

Typical readers of this book would be:

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

Prerequisites

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

Specific to the ACM ICPC Contestants

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

Specific to the IOI Contestants

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

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

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

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

Specific to the Teachers/Coaches

This book is used in Steven's CS3233 - 'Competitive Programming' course in the School of Computing, National University of Singapore. It is conducted in 13 teaching weeks using the following lesson plan (see Table 2). The PDF slides (only the public version) are given in the companion web site of this book. Hints/brief solutions of the written exercises in this book are given in Appendix A. Fellow teachers/coaches are free to modify the lesson plan to suit your students' needs.

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

Table 2: Lesson Plan

To All Readers

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

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

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

Changes for the Second Edition

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

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

- The first noticeable change is the layout. We now have more information density per page. The 2nd edition uses single line spacing instead of one half line spacing in the 1st edition. The positioning of small figures is also enhanced so that we have a more compact layout. This is to avoid increasing the number of pages by too much while we add more content.
- Some minor bug in our example codes (both the ones shown in the book and the soft copy given in the companion web site) are fixed. All example codes now have much more meaningful comments to help readers understand the code.
- Several known language issues (typo, grammatical, stylistic) have been corrected.
- On top of enhancing the writing of existing data structures, algorithms, and programming problems, we also add these *new* materials in each chapter:

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

- Many existing figures in this book are re-drawn and enhanced. Many new figures are added to help explain the concepts more clearly.
- The first edition is mainly written using ICPC and C++ viewpoint. The second edition is now written in a more balanced viewpoint between ICPC versus IOI and C++ versus Java. Java support is strongly enhanced in the second edition. However, we do not support any other programming languages as of now.
- Steven's 'Methods to Solve' website is now fully integrated in this book in form of 'one liner hints' per problem and the useful problem index at the back of this book. Now, reaching 1000 problems solved in UVa online judge is no longer a wild dream (we now consider that this feat is doable by a *serious* 4-year Computer Science university undergraduate).
- Some examples in the first edition use old programming problems. In the second edition, some examples are replaced/added with the newer examples.
- Addition of around 600 more programming exercises that Steven & Felix have solved in UVa online judge and Live Archive between the first and second edition. We also give much more conceptual exercises throughout the book with hints/short solutions as appendix.
- Addition of short profiles of data structure/algorithm inventors adapted from Wikipedia [42] or other sources. It is nice to know a little bit more about the man behind these algorithms.

Web Sites

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

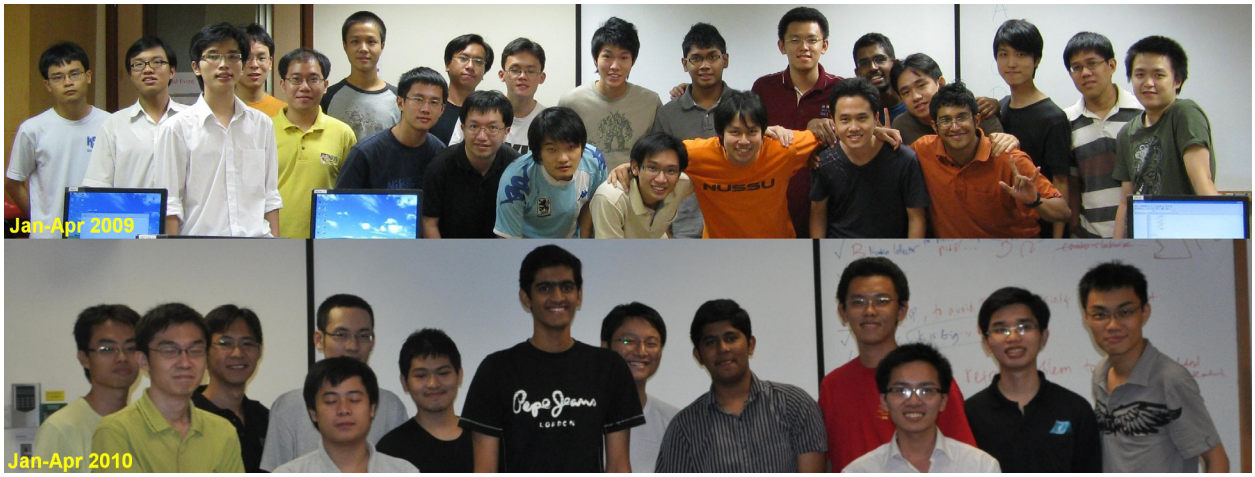
All programming exercises in this book are integrated in:
`http://felix-halim.net/uva/hunting.php`, and in
`http://uva.onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&category=118`

Acknowledgments for the First Edition

Steven wants to thank:

- God, Jesus Christ, Holy Spirit, for giving talent and passion in this competitive programming.
- My lovely wife, Grace Suryani, for allowing me to spend our precious time for this project.
- My younger brother and co-author, Felix Halim, for sharing many data structures, algorithms, and programming tricks to improve the writing of this book.
- My father Lin Tjie Fong and mother Tan Hoey Lan for raising us and encouraging us to do well in our study and work.
- School of Computing, National University of Singapore, for employing me and allowing me to teach CS3233 - 'Competitive Programming' module from which this book is born.
- NUS/ex-NUS professors/lecturers who have shaped my competitive programming and coaching skills: Prof Andrew Lim Leong Chye, Dr Tan Sun Teck, Aaron Tan Tuck Choy, Dr Sung Wing Kin, Ken, Dr Alan Cheng Holun.
- My friend Ilham Winata Kurnia for proof reading the manuscript of the first edition.
- Fellow Teaching Assistants of CS3233 and ACM ICPC Trainers @ NUS: Su Zhan, Ngo Minh Duc, Melvin Zhang Zhiyong, Bramandia Ramadhana.

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



Acknowledgments for the Second Edition

Additionally, Steven wants to thank:

- ≈ 550 buyers of the first edition as of 19 July 2011. Your supportive responses encourage us!
- Fellow Teaching Assistant of CS3233 @ NUS: Victor Loh Bo Huai.
- My CS3233 students in Sem2 AY2010/2011 who contributed in both technical and presentation aspects of the second edition of this book, in alphabetical order: Aldrian Obaja Muis, Bach Ngoc Thanh Cong, Chen Juncheng, Devendra Goyal, Fikril Bahri, Hassan Ali Askari, Harta Wijaya, Hong Dai Thanh, Koh Zi Chun, Lee Ying Cong, Peter Phandi, Raymond Hendy Susanto, Sim Wenlong Russell, Tan Hiang Tat, Tran Cong Hoang, Yuan Yuan, and one other student who prefers to be anonymous (class photo is shown below).

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

To a better future of humankind,
Steven and Felix Halim
Singapore, 19 July 2011

Authors' Profiles

Steven Halim, PhD[1]

stevenhalim@gmail.com

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

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

Felix Halim, PhD Candidate[2]

felix.halim@gmail.com

Felix Halim is currently a PhD student in the same University: SoC, NUS. In terms of programming contests, Felix has a much more colorful reputation than his older brother. He was IOI 2002 contestant (representing Indonesia). His ICPC teams (at that time, Bina Nusantara University) took part in ACM ICPC Manila Regional 2003-2004-2005 and obtained rank 10th, 6th, and 10th respectively. Then, in his final year, his team finally won ACM ICPC Kaohsiung Regional 2006 and thus became ACM ICPC World Finalist @ Tokyo 2007 (Honorable Mention). Today, he actively joins TopCoder Single Round Matches and his highest rating is a **yellow** coder.

Copyright

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

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

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

1. Given a string that represents a base X number, convert it to equivalent string in base Y, $2 \leq X, Y \leq 36$. For example: "FF" in base X = 16 (Hexadecimal) is "255" in base $Y_1 = 10$ (Decimal) and "11111111" in base $Y_2 = 2$ (binary). (More details in Section 5.3.2).
2. Given a list of integers L of size up to $1M$ items, determine whether a value v exists in L by not using more than 20 comparisons? (More details in Section 2.2.1).
3. Given a date, determine what is the day (Monday, Tuesday, ..., Sunday) of that date? (e.g. 9 August 2010 – the launch date of the first edition of this book – is Monday).
4. Given a string, replace all 'special words' of length 3 with 3 stars "***". The 'special word' starts with a lowercase alphabet character and followed by two consecutive digits, e.g.
   S = "line: a70 and z72 will be replaced, but aa24 and a872 will not"
   will be transformed to
   S = "line: *** and *** will be replaced, but aa24 and a872 will not".
5. Write the shortest possible **Java code** to read in a double
   (e.g. `1.4732`, `15.324547327`, etc)
   and print it again, but now with minimum field width 7 and 3 digits after decimal point
   (e.g. `ss1.473` (where '`s`' denotes a space), `s15.325`, etc)
6. Generate all possible permutations of {0, 1, 2, ..., N-1}, for N = 10.
7. Generate all possible subsets of {0, 1, 2, ..., N-1}, for N = 20.

1.2.4 Tip 5: Master the Art of Testing Code

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

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

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

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

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

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

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

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

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

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

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

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

     ```
     Example:            5| 4| 3| 2| 1| 0 <- 0-based indexing from right
                        32|16| 8| 4| 2| 1 <- power of 2
     S = 34 (base 10) = 1| 0| 0| 0| 1| 0 (base 2)
     ```

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

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

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

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

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

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

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

Solution: First, append two more coordinates so that `arr` = {0, the original `arr`, and L}. Then, use these Complete Search recurrences `cut(left, right)` where `left`/`right` are the left/right indices of the current stick w.r.t `arr`, respectively. Originally the stick is described by `left = 0` and `right = n+1`, i.e. a stick with length `[0..L]`:

1. `cut(i - 1, i) = 0, ∀ i ∈ [0..n-1]` // If `left+1 = right` where `left` and `right` are the indices in `arr`, then we are left with one stick segment that do not need to be cut anymore.

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

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

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

2. How do you add? (UVa 10943)

Abridged problem description: Given a number N, how many ways can K numbers less than or equal to N add up to N? Constraints: $1 \leq N, K \leq 100$. Example: For $N = 20$ and $K = 2$, there are 21 ways: $0 + 20$, $1 + 19$, $2 + 18$, $3 + 17$, ..., $20 + 0$.

Mathematically, the number of ways to add is ${}^{(n+k-1)}C_{(k-1)}$ (see Section 5.4.2 about Binomial Coefficients). Here, we will use this simple problem to re-illustrate Dynamic Programming principles that we have learned in this section.

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

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

Next, we have to determine the base case(s) of this problem. It turns out that this problem is very easy when $K = 1$. Whatever N is, there is only *one way* to add exactly one number less than or equal to N to get N: Use N itself. There is no other base case for this problem.

For the general case, we have this recursive formulation which is not too difficult to derive: At state (N, K) where $K > 1$, we can split N into one number $X \in$ `[0..N]` and $N - X$, i.e. $N = X + (N - X)$. After doing this, we have subproblem $(N - X, K - 1)$, i.e. Given a number $N - X$, how many ways can $K - 1$ numbers less than or equal to $N - X$ add up to $N - X$?

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

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

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

This problem has overlapping sub-problems[22], but there are only $N \times K$ possible states of (N, K). The cost to compute one state is $O(N)$. Thus, the overall time complexity is $O(N^2 \times K)$. As $1 \leq N, K \leq 100$, this is feasible. The answer is in `ways(N, K)`.

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

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

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

Single-Source Shortest/Longest Paths on DAG

The Single-Source Shortest Paths (SSSP) problem becomes much simpler if the given graph is a DAG. This is because a DAG has at least one topological order! We can use an $O(V+E)$ topological sort algorithm in Section 4.2.1 to find one such topological order, then relax the edges according to this order. The topological order will ensure that if we have a vertex b that has an incoming edge from a vertex a, then vertex b is relaxed *after* vertex a has obtained correct distance value. This way, the distance values propagation is correct with just one $O(V+E)$ linear pass! This is the essence of Dynamic Programming principle to avoid recomputation of overlapping subproblem to covered earlier in Section 3.5.

The Single-Source *Longest Paths* problem is a problem of finding the longest (simple[21]) paths from a starting vertex s to other vertices. The decision[22] version of this problem is NP-complete on a general graph. However the problem is again easy if the graph has no cycle, which is true in a DAG. The solution for the Longest Paths on DAG[23]is just a minor tweak from the DP solution for the SSSP on DAG shown above. One trick is to multiply all edge weights by -1 and run the same SSSP solution as above. Finally, negate the resulting values to get the actual results.

Longest Paths on DAG has applications in project scheduling (see UVa 452). We can model sub projects dependency as a DAG and the time needed to complete a sub project as vertex weight. The shortest possible time to finish the entire project is determined by the longest path (a.k.a. the *critical* path) found in this DAG. See Figure 4.28 for an example with 6 sub projects, their estimated completion time units, and their dependencies. The longest path $0 \rightarrow 1 \rightarrow 2 \rightarrow 4 \rightarrow 5$ with 16 time units determines the shortest possible time to finish the whole project. In order to achieve this, all sub projects along the longest (critical) path must be on time.

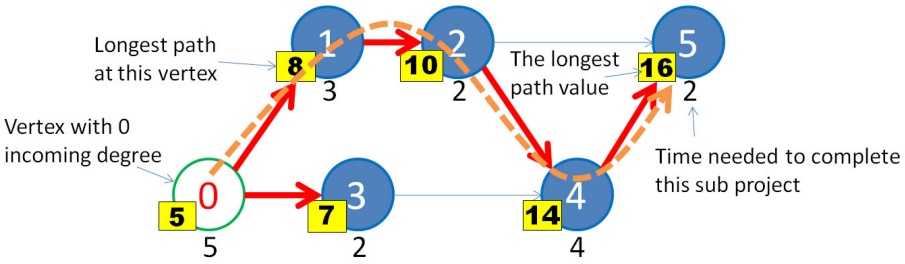


Figure 4.28: The Longest Path on this DAG is the Shortest Way to Complete the Project

Counting Paths in DAG

Motivating problem (UVa 988): In life, one has many paths to choose, leading to many different lives. Enumerate how many different lives one can live, given a specific set of choices at each point in time. One is given a list of events, and a number of choices that can be selected, for each event. The objective is to count how many ways to go from the event that started it all (birth, index 0) to an event where one has no further choices (that is, death, index n).

Clearly the problem above is a DAG as one can move forward in time, but cannot go backward. The number of such paths can be found easily with a Dynamic Programming technique on DAG. First we compute one (any) topological order (vertex 0 will always be the first and the vertex that represents death/vertex n will always be the last). We start by setting `num_paths[0] = 1`. Then, we process the remaining vertices one by one according to the topological order. When processing vertex `cur`, we update each neighbor `v` of `cur` by setting `num_paths[v] += num_paths[cur]`. After such $O(V+E)$ steps, we will know the number of paths in `num_paths[n]`. Figure 4.29 shows an example with 9 events and 6 different possible life scenarios.

[21]On general graph with positive weight edges, longest path is ill-defined as one can take a positive cycle and use that cycle to create an infinitely long path. This is the same issue as the negative cycle in shortest path problem. That is why for general graph, we use the term: 'longest *simple* path'. All paths in DAG are simple by definition.

[22]The decision version of this problem asks if the general graph has a simple path of total weight $\geq k$.

[23]The Longest Increasing Subsequence problem in Section 3.5.2 can also be modeled as Longest Paths on DAG.

- Finding Pattern or Formula
  These problems require the problem solver to read the problem description carefully to spot the pattern or simplified formula. Attacking them directly will usually result in TLE verdict. The actual solutions are usually short and do not require loops or recursions. Example: Let set S be a set of *square integers* sorted in increasing order: {1, 4, 9, 16, 25, ...} Given an integer X. How many integers in S are less than X? Answer: $\lfloor \sqrt{n} \rfloor$.

- Grid
  These problems involve grid manipulation. The format of the grid can be complex, but the grid will follow some primitive rules. The 'trivial' 1D/2D grid are not classified here. The solution usually depends on the problem solver's creativity on finding the patterns to manipulate/navigate the grid or in converting the given one into a simpler one.

- Number Systems or Sequences
  Some Ad Hoc mathematics problems involve definitions of existing (or fictional) **Number Systems or Sequences** and our task is to produce either the number (sequence) within some range or the n-th one, verify if the given number (sequence) is valid according to definition, etc. Usually, following the problem description carefully is the key to solving the problem.

- Logarithm, Exponentiation, Power
  These problems involve the (clever) usage of `log()` and/or `exp()` function.
  Exercise 5.2.1: What should we use in C/C++/Java to compute $\log_b$`(a)` (base b)?
  Exercise 5.2.2: What will be returned by `(int)floor(1 + log10((double)a))`?
  Exercise 5.2.3: How to compute $\sqrt[n]{a}$ (the n-th root of a) in C/C++/Java?

- Polynomial
  These problems involve polynomial: evaluation, derivation, multiplication, division, etc.
  We can represent a polynomial by storing the coefficients of the polynomial's terms sorted by their powers. Usually, the operations on polynomial require some careful usage of loops.

- Base Number Variants
  These are the mathematical problems involving base number, but they are not the *standard* conversion problem that can be easily solved with Java BigInteger technique (see Section 5.3).

- Just Ad Hoc
  These are other mathematics-related problems that cannot be classified as one of the above.

We suggest that the readers – especially those who are new with mathematics-related problems – to kick start their training programme on solving mathematics-related problems by solving at least 2 or 3 problems *from each sub-category*, especially the ones that we highlight as **must try ***. Note: The problems listed below constitute ≈ 30% of the entire problems in this chapter.

Programming Exercises related to Ad Hoc Mathematics problems:

- The Simpler Ones
  1. UVa 10055 - Hashmat the Brave Warrior (absolute function; use long long)
  2. UVa 10071 - Back to High School Physics (super simple: outputs $2 \times v \times t$)
  3. UVa 10281 - Average Speed (distance = speed × time elapsed)
  4. UVa 10469 - To Carry or not to Carry (super simple if you use `xor`)
  5. **UVa 10773 - Back to Intermediate Math *** (several tricky cases)
  6. UVa 11614 - Etruscan Warriors Never ... (find roots of a quadratic equation)
  7. **UVa 11723 - Numbering Road *** (simple math)
  8. UVa 11805 - Bafana Bafana (very simple $O(1)$ formula exists)
  9. **UVa 11875 - Brick Game *** (get median of a sorted input)

6.4 String Matching

String *Matching* (a.k.a String *Searching*[4]) is a problem of finding the starting index (or indices) of a (sub)string (called *pattern* `P`) in a longer string (called *text* `T`). For example, let's assume that we have `T = 'STEVEN EVENT'`. If `P = 'EVE'`, then the answers are index 2 and 7 (0-based indexing). If `P = 'EVENT'`, then the answer is index 7 only. If `P = 'EVENING'`, then there is no answer (no matching found and usually we return either -1 or NULL).

6.4.1 Library Solution

For most *pure* String Matching problems on reasonably short strings, we can just use string library in our programming language. It is `strstr` in C <cstring>, `find` in C++ <string>, `indexOf` in Java `String` class. We shall not elaborate further and let you explore these library functions by solving some of the programming exercises mentioned at the end of this section.

6.4.2 Knuth-Morris-Pratt (KMP) Algorithm

In Section 1.2.2, Question 7, we have an exercise of finding all the occurrences of a substring P (of length m) in a (long) string T (of length n), if any. The code snippet, reproduced below, is actually the *naïve* implementation of String Matching algorithm.

```
void naiveMatching() {
  for (int i = 0; i < n; i++) {            // try all potential starting indices
    bool found = true;
    for (int j = 0; j < m && found; j++)             // use boolean flag 'found'
      if (P[j] != T[i + j])                                 // if mismatch found
        found = false;               // abort this, shift starting index i by +1
    if (found)                                      // if P[0..m-1] == T[i..i+m-1]
      printf("P is found at index %d in T\n", i);
} }
```

This naïve algorithm can run in $O(n)$ *on average* if applied to natural text like the paragraphs of this book, but it can run in $O(nm)$ with the worst case programming contest input like this: `T = 'AAAAAAAAAAB'` ('A' ten times and then one 'B') and `P = 'AAAAB'`. The naive algorithm will keep failing at the last character of pattern `P` and then try the next starting index which is just +1 than the previous attempt. This is not efficient.

In 1977, Knuth, Morris, and Pratt – thus the name of KMP – invented a better String Matching algorithm that makes use of the information gained by previous character comparisons, especially those that matches. KMP algorithm *never* re-compares a character in `T` that has matched a character in `P`. However, it works similar to the naïve algorithm if the *first* character of pattern `P` and the current character in `T` is a mismatch. In the example below[5], comparing `T[i + j]` and `P[j]` from `i` = 0 to 13 with `j = 0` (first character) is no different than the naïve algorithm.

```
              1         2         3         4         5
    01234567890123456789012345678901234567890123456789012345678901234567890
T = I DO NOT LIKE SEVENTY SEV BUT SEVENTY SEVENTY SEVEN
P = SEVENTY SEVEN
    0123456789012
              1
    ^ the first character of P always mismatch with T[i] from index i = 0 to 13
    KMP has no choice but to shift starting index i by +1, as with naive matching.
```

[4]We deal with this String Matching problem almost every time we read/edit text using computer. How many times have you pressed the well-known 'CTRL + F' button (standard Windows shortcut for the 'find feature') in typical word processing softwares, web browsers, etc.

[5]The sentence in string `T` below is just for illustration. It is not grammatically correct.

Chapter 7

(Computational) Geometry

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

7.1 Overview and Motivation

(Computational[1]) Geometry is yet another topic that frequently appears in programming contests. Almost all ICPC problem sets have *at least one* geometry problem. If you are lucky, it will ask you for some geometry solution that you have learned before. Usually you draw the geometrical object(s) and then derive the solution from some basic geometric formulas. However, many geometry problems are the *computational* ones that require some complex algorithm(s).

In IOI, the existence of geometry-specific problems depends on the tasks chosen by the Scientific Committee that year. In recent years (2009 and 2010), IOI tasks do not feature geometry problems. However, in earlier years, there exists one or two geometry related problems per IOI [39].

We have observed that many contestants, especially ICPC contestants, are 'afraid' to tackle geometry-related problems due to two logical reasons:

1. The solutions for geometry-related problems have *lower* probability of getting Accepted (AC) during contest time compared to the solutions for other problem types in the problem set, i.e. Complete Search or Dynamic Programming problems. This make attempting *other* problem types in the problem set more worthwhile than spending precious minutes coding geometry solution that has lower probability of acceptance.

    - There are usually several tricky 'special corner cases' in geometry problems, e.g. What if the lines are vertical (infinite gradient)?, What if the points are collinear?, What if the polygon is concave?, etc. It is usually a very good idea to test your team's geometry solution with lots of test cases before you submit it for judging.
    - There is a possibility of having floating point precision errors that cause even a 'correct' algorithm to get a Wrong Answer (WA) response.

2. The contestants are not well prepared.

    - The contestants forget some important basic formulas or unable to derive the required formulas from the basic ones.
    - The contestants do not prepare well-written library functions and their attempts to code such functions during stressful contest time end up with (lots of) bugs. In ICPC, the top teams usually fill sizeable part of their hard copy material (which they can bring into the contest room) with lots of geometry formulas and library functions.

[1]We differentiate between *pure* geometry problems and the *computational* geometry ones. Pure geometry problems can normally be solved by hand (pen and paper method). Computational geometry problems typically require running an algorithm using computer to obtain the solution.

graph is non Eulerian, e.g. see the graph in Figure 8.3 (left), then this Chinese Postman Problem is harder.

The important insight to solve this problem is to realize that a non Eulerian graph G must have an *even number* of vertices of odd degree. This is an observation found by Euler himself. Let's name the subset of vertices of G that have odd degree as T. Now, create a complete graph K_n where n is the size of T. T form the vertices of K_n. An edge (i, j) in K_n has weight according to the weight of the original edge (if it exists), or the shortest path weight of a path from i to j, e.g. edge `2-5` has weight $2 + 1 = 3$ from path `2-4-5`.

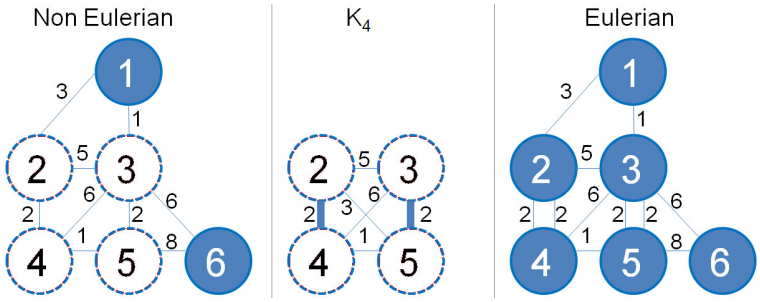


Figure 8.3: An Example of Chinese Postman Problem

Now, if we *double* the edges selected by the *minimum weight perfect matching* on this complete graph K_n, we will convert the non Eulerian graph G to another graph G' which is Eulerian. This is because by doubling those edges, we actually add an edge between a pair of vertices with odd degree (thus making them have even degree afterwards). The *minimum weight* perfect matching ensures that this transformation is done in the *least cost way*. The solution for the minimum weight perfect matching on the K_4 shown in Figure 8.3 (middle) is to take edge `2-4` (with weight 2) and edge `3-5` (also with weight 2).

After doubling edge `2-4` and edge `3-5`, we are now back to the easy case of the Chinese Postman Problem. In Figure 8.3 (right), we have an Eulerian graph. The Chinese Postman problem tour is simple in this Eulerian graph. One such tour is: `1->2->4->5->3->2->4->3->5->6->3->1` with total weight of `38` (the sum of all edge weight in the modified Eulerian graph).

The hardest part of solving the Chinese Postman Problem is in finding the minimum weight perfect matching on K_n. If n is small, this part can be solved with DP + bitmask technique shown in Section 8.4.1 above. Otherwise, we have to use Edmonds' Blossom algorithm [6].

8.4.3 Compilation of Common DP States

After solving lots of DP problems, contestants will develop a sense of which parameters are commonly selected to represent distinct states of a DP problem. Some of them are as follows:

1. Current position in an ordered array
   Original problem: $[x_1, x_2, \ldots, x_n]$,
   e.g. a list/sequence of numbers (integer/double), a string (character array)
   Sub problems: Break the original problem into:

   - Sub problem and suffix: $[x_1, x_2, \ldots, x_{n-1}] + x_n$
   - Prefix and sub problem: $x_1 + [x_2, x_3, \ldots, x_n]$
   - Two sub problems: $[x_1, x_2, \ldots, x_i] + [x_{i+1}, x_{i+2}, \ldots, x_n]$

   Example: LIS, 1D Max Sum, Matrix Chain Multiplication (MCM), etc (see Section 3.5.2)

2. Current positions in two ordered arrays (similar as above, but on two arrays)
   Original problem: $[x_1, x_2, \ldots, x_n]$ and $[y_1, y_2, \ldots, y_n]$,
   e.g. two lists/sequences/strings
   Sub problems: Break the original problem into:

```
// C++ code for question 7, assuming all necessary includes have been done
int main() {
  int p[20], N = 20;
  for (int i = 0; i < N; i++) p[i] = i;
  for (int i = 0; i < (1 << N); i++) {
    for (int j = 0; j < N; j++)
      if (i & (1 << j)) // if bit j is on
        printf("%d ", p[j]);                                  // this is part of set
    printf("\n");
} }
```

Exercise 1.2.4: Answers for situation judging are in bracket:

1. You receive a WA response for a very easy problem. What should you do?
   (a) Abandon this problem and do another. **(not ok, your team will lose out).**
   (b) Improve the performance of your solution. **(not useful).**
   (c) Create tricky test cases and find the bug. **(the most logical answer).**
   (d) (In ICPC): Ask another coder in your team to re-do this problem. **(this is a logical answer. this can work although your team will lose precious penalty time).**

2. You receive a TLE response for an your $O(N^3)$ solution. However, maximum N is just 100. What should you do?
   (a) Abandon this problem and do another. **(not ok, your team will lose out).**
   (b) Improve the performance of your solution. **(not ok, we should not get TLE with an $O(N^3)$ algorithm if $N \leq \approx 200$).**
   (c) Create tricky test cases and find the bug. **(this is the answer; maybe your program is accidentally trapped in an infinite loop in some test cases).**

3. Follow up question (see question 2 above): What if maximum N is 100.000?
   (If $N > 200$, you have no choice but to improve the performance of the algorithm or use a faster algorithm).

4. You receive an RTE response. Your code runs OK in your machine. What should you do?
   Possible causes for RTE are usually array size too small or stack overflow/infinite recursion. Design test cases that can possibly cause your code to end up with these situations.

5. One hour to go before the end of the contest. You have 1 WA code and 1 fresh idea for *another* problem. What should you (your team) do?
   (a) Abandon the problem with WA code, switch to that other problem in attempt to solve one more problem. **(in individual contests like IOI, this may be a good idea).**
   (b) Insist that you have to debug the WA code. There is not enough time to start working on a new code. **(if the idea for another problem involves complex and tedious code, then deciding to focus on the WA code may be a good idea rather than having two incomplete/'non AC' codes).**
   (c) (In ICPC): Print the WA code. Ask two other team members to scrutinize the printed code while one coder switches to that other problem in attempt to solve TWO more problems. **(if the idea for another problem is can be coded in less than 30 minutes, then code this one while hoping your team mates can find the bug for the WA code by looking at the printed code).**

Appendix B

uHunt

uHunt (`http://felix-halim.net/uva/hunting.php`) is a web-based tool created by one of the author of this book (Felix Halim) to complement the UVa online judge [28]. This tool is created to help UVa users to keep track which problems that he/she has (and has not) solved. Considering that UVa has ≈ 2950 problems, today's programmers indeed need a tool like this (see Figure B.1).

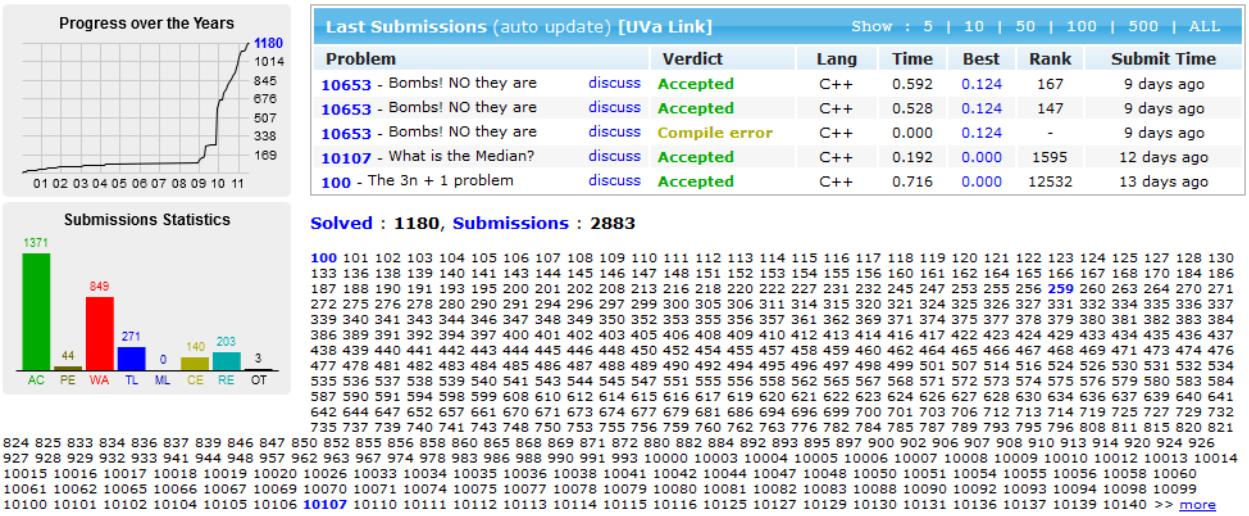
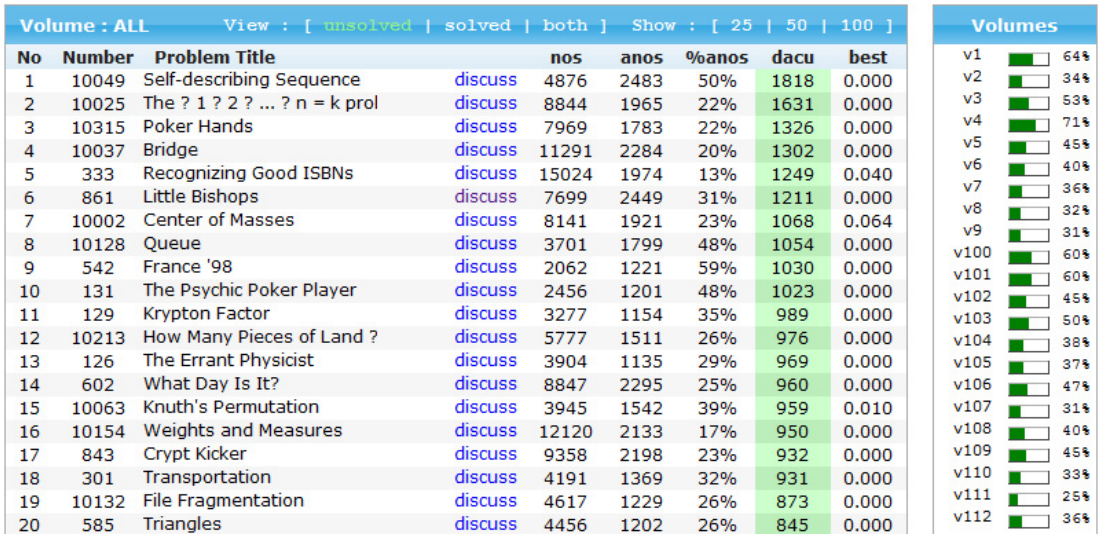


Figure B.1: Steven's statistics as of 19 July 2011

The original version of uHunt (that stands for UVa hunting) is to help UVa users in finding the 20 easiest problems to solve next. The rationale is this: If a user is still a beginner and he/she needs to build up his/her confidence, he/she needs to solve problems with gradual difficulty. This is much better than directly attempting hard problems and keep getting non Accepted (AC) responses without knowing what's wrong. The ≈ 114744 UVa users actually contribute statistical information for each problem that can be exploited for this purpose. The easier problems will have higher number of submissions and higher number of AC. However, as a UVa user can still submit codes to a problem even though he/she already gets AC for that problem, then the number of AC alone is not an accurate measure to tell whether a problem is easy or not. An extreme example is like this: Suppose there is a hard problem that is attempted by a single good programmer who submits 50 AC codes just to improve his code's runtime. This problem is not easier than another easier problem where only 49 different users get AC. To deal with this, the default sorting criteria in uHunt is 'dacu' that stands for 'distinct accepted users'. The hard problem in the extreme example above only has dacu = 1 whereas the easier problem has dacu = 49 (see Figure B.2).

Today's uHunt is much more flexible than its predecessors as it also allows users to rank problems that they *already solve* based on the runtime difference of their AC code versus the best AC code. A (wide) gap implies that the user still does not know a certain algorithms, data structures, or hacking tricks to get that faster performance. uHunt also has the 'statistics comparer' feature. If you have a *rival* (or a better UVa user that you admire), you can compare your list of solved problems with him/her and then try to solve the problems that your rival can solve.



Volume : ALL View : [unsolved | solved | both] Show : [25 | 50 | 100]

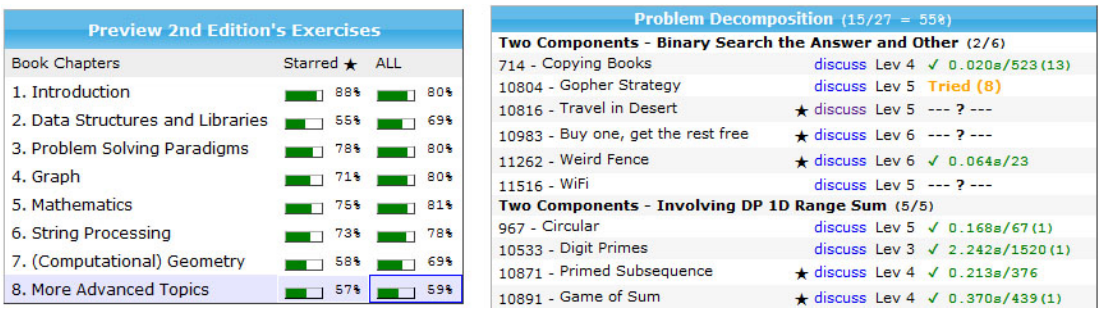
No	Number	Problem Title		nos	anos	%anos	dacu	best
1	10049	Self-describing Sequence	discuss	4876	2483	50%	1818	0.000
2	10025	The ? 1 ? 2 ? ... ? n = k prol	discuss	8844	1965	22%	1631	0.000
3	10315	Poker Hands	discuss	7969	1783	22%	1326	0.000
4	10037	Bridge	discuss	11291	2284	20%	1302	0.000
5	333	Recognizing Good ISBNs	discuss	15024	1974	13%	1249	0.040
6	861	Little Bishops	discuss	7699	2449	31%	1211	0.000
7	10002	Center of Masses	discuss	8141	1921	23%	1068	0.064
8	10128	Queue	discuss	3701	1799	48%	1054	0.000
9	542	France '98	discuss	2062	1221	59%	1030	0.000
10	131	The Psychic Poker Player	discuss	2456	1201	48%	1023	0.000
11	129	Krypton Factor	discuss	3277	1154	35%	989	0.000
12	10213	How Many Pieces of Land ?	discuss	5777	1511	26%	976	0.000
13	126	The Errant Physicist	discuss	3904	1135	29%	969	0.000
14	602	What Day Is It?	discuss	8847	2295	25%	960	0.000
15	10063	Knuth's Permutation	discuss	3945	1542	39%	959	0.010
16	10154	Weights and Measures	discuss	12120	2133	17%	950	0.000
17	843	Crypt Kicker	discuss	9358	2198	23%	932	0.000
18	301	Transportation	discuss	4191	1369	32%	931	0.000
19	10132	File Fragmentation	discuss	4617	1229	26%	873	0.000
20	585	Triangles	discuss	4456	1202	26%	845	0.000

Volumes

v1 64% v2 34% v3 53% v4 71% v5 45% v6 40% v7 36% v8 32% v9 31% v100 60% v101 60% v102 45% v103 50% v104 38% v105 37% v106 47% v107 31% v108 40% v109 45% v110 33% v111 25% v112 36%

Figure B.2: Hunting the next easiest problems using 'dacu'

Another new feature since 2010 is the integration of the ≈ 1198 programming exercises from this book (see Figure B.3). Now, a user can customize his/her training programme to solve *problems of similar type*! Without such (manual) categorization, this training mode is hard to execute. We also give stars (*) to problems that we consider as **must try *** (up to 3 problems per category).



Preview 2nd Edition's Exercises

Book Chapters	Starred ★	ALL
1. Introduction	88%	80%
2. Data Structures and Libraries	55%	69%
3. Problem Solving Paradigms	78%	80%
4. Graph	71%	80%
5. Mathematics	75%	81%
6. String Processing	73%	78%
7. (Computational) Geometry	58%	69%
8. More Advanced Topics	57%	59%

Problem Decomposition (15/27 = 55%)

Two Components - Binary Search the Answer and Other (2/6)

714 - Copying Books	discuss	Lev 4	✓ 0.020s/523(13)
10804 - Gopher Strategy	discuss	Lev 5	Tried (8)
10816 - Travel in Desert	★ discuss	Lev 5	--- ? ---
10983 - Buy one, get the rest free	★ discuss	Lev 6	--- ? ---
11262 - Weird Fence	★ discuss	Lev 6	✓ 0.064s/23
11516 - WiFi	discuss	Lev 5	--- ? ---

Two Components - Involving DP 1D Range Sum (5/5)

967 - Circular	discuss	Lev 5	✓ 0.168s/67(1)
10533 - Digit Primes	discuss	Lev 3	✓ 2.242s/1520(1)
10871 - Primed Subsequence	★ discuss	Lev 4	✓ 0.213s/376
10891 - Game of Sum	★ discuss	Lev 4	✓ 0.370s/439(1)

Figure B.3: The programming exercises in this book are integrated in uHunt

Building a web-based tool like uHunt is a computational challenge. There are over ≈ 9038508 submissions from ≈ 114744 users (≈ one submission every few seconds). The statistics and rankings must be updated frequently and such update must be fast. To deal with this challenge, Felix uses lots of advanced data structures (some are beyond this book), e.g. database cracking [16], Fenwick Tree, data compression, etc.

We ourselves are using this tool extensively, as can be seen in Figure B.4. Two major milestones that can be seen from our progress chart are: Felix's intensive training to eventually won ACM ICPC Kaohsiung 2006 with his ICPC team and Steven's intensive problem solving activity in the past two years (late 2009-present) to prepare this book.

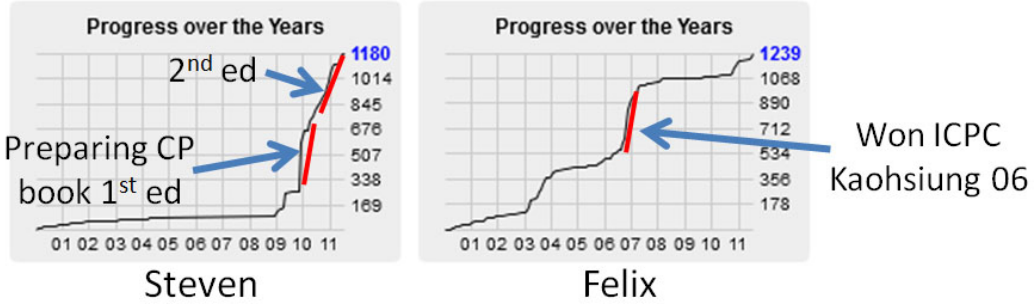


Figure B.4: Steven's & Felix's progress in UVa online judge (2000-present)

Appendix C

Credits

The problems discussed in this book are mainly taken from UVa online judge [28], ACM ICPC Live Archive [20], and past IOI tasks (mainly from 2009-2010). So far, we have contacted the following authors to get their permissions (in alphabetical order): Brian C. Dean, Colin Tan Keng Yan, Derek Kisman, Gordon V. Cormack, Jane Alam Jan, Jim Knisely, Jittat Fakcharoenphol, Manzurur Rahman Khan, Melvin Zhang Zhiyong, Michal Forisek, Mohammad Mahmudur Rahman, Piotr Rudnicki, Rob Kolstad, Rujia Liu, Shahriar Manzoor, Sohel Hafiz, and TopCoder, Inc (for PrimePairs problem in Section 4.7.4). A compilation of photos with some of these problem authors that we managed to meet in person is shown below.

However, due to the fact that there are thousands ($\approx$ 1198) of problems listed and discussed in this book, there are many problem authors that we have not manage to contact yet. If you are those problem authors or know the person whose problems are used in this book, please notify us. We keep a more updated copy of this problem credits in our supporting website:
`https://sites.google.com/site/stevenhalim/home/credits`

Appendix D

Plan for the Third Edition

The first edition of this book is released on 9 August 2010 and the second edition one year later on 1 August 2011. Is this going to be a yearly trend?

We all have to admit that the world of competitive programming is constantly evolving and the programming contest problems are getting harder. To avoid getting *too* outdated, it is natural for a book like this one to have incremental updates (anyway, a book is always out of date by the time it goes to print). Since we believe that a good book is the one that keeps being updated by it's authors, we do have plan to release the third edition in the future, but not in 2012.

The gap between the first and the second edition is 'small' (only one year) as there are so much to write if one writes a book from scratch. We believe that the second edition is already a significant improvement over the first edition. This makes the expectation for the third edition even higher. To have a good quality third edition, we plan to release the third edition only in two or three years time (that means around 2013 or 2014). So, the second edition edition of this book has about two to three years active life in the competitive programming world.

These are our big plans to prepare the third edition in two or three years time:

- Deal with the errata (hopefully minimal) that are found after the release of second edition. Online errata is at: `https://sites.google.com/site/stevenhalim/home/third`
- We will keep improving our problem solving skills and solve many more programming problems. By the time we are ready to publish the third edition, we expect the number of problems that we (Steven and Felix) have solved (combined) is around 2000, instead of 1502 as of now ($\approx$ 1198 of which are discussed). We will then share what we learn from those problems.
- Make the existing Chapter 1-7 a little bit more user friendly to reach newer competitive programmers: more examples, more illustrations, more conceptual exercises.
- Add discussion of most data structures and algorithms that are currently only mentioned in the chapter notes.
- Expand Chapter 8 (substantially) to serve the needs of today's competitive programmers who are already (or trying to) master the content of the second edition of this book.

We need your feedbacks

You, the reader, can help us improve the quality of the third edition of this book. If you spot any technical, grammatical, spelling errors, etc in this book or if you want to contribute certain parts for the third edition of this book (i.e. I have a better example/algorithm to illustrate a certain point), etc, please send an email to the main author directly: stevenhalim@gmail.com.

Bibliography

[1] Ahmed Shamsul Arefin. *Art of Programming Contest (from Steven's old Website)*. Gyankosh Prokashoni (Available Online), 2006.

[2] Frank Carrano. *Data Abstraction & Problem Solving with C++*. Pearson, 5th edition, 2006.

[3] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Cliff Stein. *Introduction to Algorithm*. MIT Press, 2nd edition, 2001.

[4] Sanjoy Dasgupta, Christos Papadimitriou, and U Vazirani. *Algorithms*. McGraw Hill, 2008.

[5] Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Cheong Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer, 2nd edition, 2000.

[6] Jack Edmonds. Paths, trees, and flowers. *Canadian Journal on Maths*, 17:449–467, 1965.

[7] Fabian Ernst, Jeroen Moelands, and Seppo Pieterse. Teamwork in Prog Contests: 3 * 1 = 4. http://xrds.acm.org/article.cfm?aid=332139.

[8] Project Euler. Project Euler. http://projecteuler.net/.

[9] Peter M. Fenwick. A New Data Structure for Cumulative Frequency Tables. *Software: Practice and Experience*, 24 (3):327–336, 1994.

[10] Michal Forišek. IOI Syllabus. http://people.ksp.sk/ misof/ioi-syllabus/ioi-syllabus-2009.pdf.

[11] Michal Forišek. The difficulty of programming contests increases. In *International Conference on Informatics in Secondary Schools*, 2010.

[12] Felix Halim, Roland Hock Chuan Yap, and Yongzheng Wu. A MapReduce-Based Maximum-Flow Algorithm for Large Small-World Network Graphs. In *ICDCS*, 2011.

[13] Steven Halim and Felix Halim. Competitive Programming in National University of Singapore. Ediciones Sello Editorial S.L. (Presented at Collaborative Learning Initiative Symposium CLIS @ ACM ICPC World Final 2010, Harbin, China, 2010.

[14] Steven Halim, Roland Hock Chuan Yap, and Felix Halim. Engineering SLS for the Low Autocorrelation Binary Sequence Problem. In *Constraint Programming*, pages 640–645, 2008.

[15] Steven Halim, Roland Hock Chuan Yap, and Hoong Chuin Lau. An Integrated White+Black Box Approach for Designing & Tuning SLS. In *Constraint Programming*, pages 332–347, 2007.

[16] Stratos Idreos. *Database Cracking: Towards Auto-tuning Database Kernels*. PhD thesis, CWI and University of Amsterdam, 2010.

[17] TopCoder Inc. Algorithm Tutorials. http://www.topcoder.com/tc?d1=tutorials&d2=alg_index&module=Static.

Index