Book 1

Chapter 1-4

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

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 is 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.

---

[1] Please 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.

L-R: Fredrik Niemelä, Carlos, Miguel Revilla, Miguel Jr, Felix, Steven

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 common-place 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 provide 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 `typedef`s 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> vii;                  // 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 gap[2] 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.

---

[2]Including 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 tool[3] 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 `pbds` 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 most[4] of the IOI syllabus [16] as of year 2020.

---

[3]VisuAlgo 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.

[4]Note 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

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

Authors' Profiles

Steven Halim, PhD[5]

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 (RRwatameda), 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	RRwatameda	Joint-14/128	2016
Ekaterinburg, Russia	ThanQ+	Joint-19/122	2014
Rapid City, USA	TeamTam	Joint-20/133	2017

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

---

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

Felix Halim, PhD[6]

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[7]

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.

---

[6]PhD Thesis: "Solving Big Data Problems: from Sequences to Tables and Graphs", 2012.
[7]PhD Thesis: "Graph Properties and Algorithms in Social Networks: Privacy, Sybil Attacks, and the Computer Science Community", 2017.

Abbreviations

A* : A Star
ACM : Assoc for Computing Machinery
AC : Accepted
ADT : Abstract Data Type
AL : Adjacency List
AM : Adjacency Matrix
APSP : All-Pairs Shortest Paths
AVL : Adelson-Velskii Landis (BST)

BNF : Backus Naur Form
BFS : Breadth First Search
BI : Big Integer
BIT : Binary Indexed Tree
bBST : (balanced) Binary Search Tree
BSTA : Binary Search the Answer

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

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

ED : Edit Distance
EL : Edge List

FFT : Fast Fourier Transform
FIFO : First In First Out
FT : Fenwick Tree

GCD : Greatest Common Divisor

HLD : Heavy-Light Decomposition

ICPC : Intl. Collegiate Prog. Contest
IDS : Iterative Deepening Search
IDA* : Iterative Deepening A Star

I/O : Input/Output
IOI : Intl. Olympiad in Informatics
IPSC : Internet Problem Solving Contest

KISS : Keep It Short and Simple

LA : Live Archive [30]
LCA : Lowest Common Ancestor
LCE : Longest Common Extension
LCM : Least Common Multiple
LCP : Longest Common Prefix
LCS$_1$: Longest Common Subsequence
LCS$_2$: Longest Common Substring
LIFO : Last In First Out
LIS : Longest Increasing Subsequence
LRS : Longest Repeated Substring
LSB : Least Significant Bit

MCBM : Max Cardinality Bip. Matching
MCM$_1$: Max Cardinality Matching
MCM$_2$: Matrix Chain Multiplication
MCMF : Min-Cost Max-Flow
MIS : Max Independent Set
MLE : Memory Limit Exceeded
MPC : Min Path Cover
MSB : Most Significant Bit
MSSP : Multi-Sources Shortest Paths
MST : Min Spanning Tree
MWIS : Max Weight Independent Set
MVC : Min Vertex Cover
MWVC : Min Weight Vertex Cover

NP : Non-deterministic Polynomial

OJ : Online Judge

PE : Presentation Error

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

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

1.1 Competitive Programming

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

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

An illustration: UVa Online Judge [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 $cost = \sum_{i=1}^{N} d_i$ is **minimized**. Output the minimum *cost* as a floating point number with 2 digits precision in one line. Constraints: $1 \le N \le 8$ and $0 \le x, y \le 1000$.

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):
$cost = 4.83$.

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

---

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

[2]By hiding the actual test data from the problem statement, competitive programming encourages the problem solvers to exercise their mental strength to think of 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.
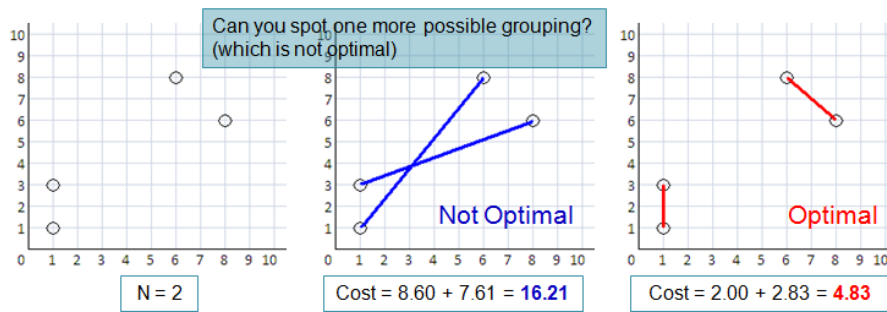
Figure 1.1: Illustration of UVa 10911 - Forming Quiz Teams

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 hosts[3] 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 practice[4] session and two contest days[5], 5 hours per session. IOI is an individual contest[6]. 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.

---

[3]Online IOI in 2020 due to COVID-19.
[4]The problems are usually already distributed online a few weeks prior to the actual IOI.
[5]It is important to maintain stamina and emotional well-being for *both* contest days.
[6]IOI 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 ties[7] (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) subtasks[8].

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 degree[9]. 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

ICPC[10] 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.

---

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

[8]Implementing 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.

[9]Many are with scholarships.

[10]ICPC 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 penalty[11] 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.

---

[11]Some 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 you[12]!

In the subsequent chapters, you will learn many things from the basic to the intermediate or even to the advanced[13] 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 minutes[14] 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 wpm[15], 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.

---

[12]Notice 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.

[13]Whether 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.

[14]Fast 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.

[15]A 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 possible[16]:

```cpp
/*          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)              // build distance table
      for (int j = i+1; j < 2*N; ++j)            // use 'hypot' function
        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.

---

[16]Notice 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 ($\approx$ 10-13 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 days[17] 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)

---

[17]In 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 process[19] 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 important[20] 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 ICPCs), 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:

---

[18]Discussion: 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!

[19]Treat 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.

[20]If 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 a only a rough upper bound. The actual complexity depends on what actions are done per level and whether pruning is possible.
  - A Dynamic Programming algorithm or other iterative routine which processes a 2D $n \times n$ matrix in $O(k)$ per cell runs in $O(k \times n^2)$ time. This is explained in further detail in Section 3.5.
  - 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 Integer[21] (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.

---

[21]`gcc` 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 contestants[22]: 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 second[23] (the typical time limit in most UVa/Kattis problems [44, 34]), we can predict the 'worst' algorithm that can still pass the time limit[24]. 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 sizes[25]. 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.

---

[22]Unlike ICPC, the IOI tasks can usually be solved (partially or fully) using several possible solutions, each with different time complexities and subtask scores. To gain valuable points, it may be good to initialy 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.

[23]In 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*.

[24]Try problem *Kattis - tutorial **.

[25]It 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 `sum(i, j)`, i.e., the sum of `L[i] + L[i+1] + ...+ 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 `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)

   (a) Sieve of Eratosthenes.

   (b) $\forall i \in$ `[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? $1 \leq n \leq 10^{14}$.

   (a) Test if $n! \% m == 0$.

   (b) 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?

   (a) 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);
   }
   ```

   (b) There are better algorithms, we can use: _____ (Book 2).

9. Given a set S of N points scattered on a 2D plane ($2 \leq N \leq 5\,000$), find two points $\in S$ that have the greatest separating Euclidean distance. Is an $O(N^2)$ complete search algorithm that tries all possible pairs feasible?

   (a) Yes, such complete search is possible.

   (b) No, we must find another way. We must use: _____ .

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.

    (a) The $O(N^2)$ complete search can still be used.

    (b) 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.

    (a) The $O(n^2)$ complete search can still be used.

    (b) The better solution using algorithm in Book 2 can still be used.

    (c) We need to use: _____

1.3.4 Tip 4: Master Programming Languages

There are several programming languages supported in ICPC[26], 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.

---

[26][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...
0.73412312234443443899923899277...
```

One possible solution is as follows:

```cpp
#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[27].

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 2×3 matrix. For this input, we should output $A' = [(1, 4), (2, 5), (3, 6)]$, i.e., a 3×2 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:

```python
A = [(1, 2, 3), (4, 5, 6)]          # list A = 2 tuples of 3
[*zip(*A)]                          # [(1, 4), (2, 5), (3, 6)]
```

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: ch1/factorial.py|java; ch1/scanf.cpp|ml; ch1/zip.py

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.

---

[27]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 π to n digits after the decimal point (rounded).
   (e.g., for $n = 2$, print `3.14`; for $n = 4$, print `3.1416`; for $n = 5$, prints `3.14159`.)

3. Given a date (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)[28], 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[29]. 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.

---

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

[29][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[30] (`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 ≥ 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 ≈ 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/`).
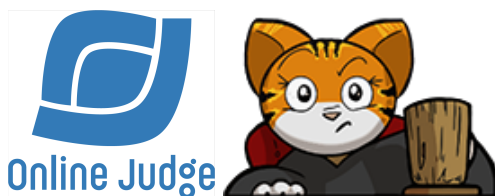
Figure 1.2: Left: (UVa) Online Judge; Right: Kattis

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 ≥ 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

---

[30]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 uses the computer for other problem). 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[31] 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 (≤ 100) in one line, output their sum in one line. We will illustrate three[32] 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.

---

[31]Kattis online judge [34] discourages this and prefers problem authors to specify one test case per file.
[32]Note that this list is not exhaustive!

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

Case Numbers and Blank Lines

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

```
C/C++ Source Code                          | Sample Input | Sample Output
------------------------------------------------------------------------
int a, b, c = 0;                           | 1 2          | Case 1: 3
while (scanf("%d %d", &a, &b) != EOF)      | 5 7          |
  // notice the two '\n'                   | 6 3          | Case 2: 12
  printf("Case %d: %d\n\n", ++c, a+b);     |-------------|
                                           |              | Case 3: 9
                                           |              |
                                           |              |-------------
```

Some other problems require us to output blank lines only *between* test cases. If we use the approach above, we will end up with an extra new line at the end of our output, producing an unnecessary 'Presentation Error' (PE) verdict[33]. We should use the following code:

---

[33]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/IO.cpp|java|py|ml

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 ≈ 4965 problems in UVa online judge [44] and another ≈ 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[34] I/O format. But from here onwards, the categorized problems use a mix of basic programming methodology techniques listed above.

---

[34]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 ($0 \le a, b, c \le 10$, $1 \le n \le 20$), output 'YES' if $a > 1$, $b > 1$, $c > 1$, and $a + b + c \ge n$, 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.

```c
#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 $1 \le n < 3$. Notice that the problem author cunningly wrote $1 \le n \le 20$ 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.

```cpp
#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)) ?
         "YES\n" : "NO\n");                        // use ternary operator
  return 0;                                        // for shorter code
}
```

Source code: `ch1/moscowdream.cpp|java|py|ml`

Programming Exercises to get you started[35]:

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*.

---

[35]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[36])

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 ≥ 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).

---

[36]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, 10141, 10324, 11586, 11661, 12614, 13007.*

   Extra Kattis: *boundingrobots*, *climbingstairs*, *deathtaxes*, *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 (freopen, 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++[37], Python[38], 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 looooooooooooong 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}.

---

[37]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.

[38]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[39] 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`

---

[39]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, $\approx$ 1-2 problems out of every $\approx$ 10-13 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[40], 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/$\diamondsuit$, C/Club/$\clubsuit$, H/Heart/$\heartsuit$, and S/Spades/$\spadesuit$) and ranks (usually: $2 < 3 < \ldots < 9 < $ T/Ten $<$ J/Jack $<$ Q/Queen $<$ K/King $<$ A/Ace[41]). It may be a good idea to map these troublesome strings to integer indices. For example, one possible mapping is to map D2 $\to$ 0, D3 $\to$ 1, ..., DA $\to$ 12, C2 $\to$ 13, C3 $\to$ 14, ..., SA $\to$ 51. Then, you can work with the integer indices instead.

---

[40]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.

[41]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 8×8 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[42], 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[43] 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`

---

[42]Knowing the details of these games can sometimes be detrimental if the rules of the game are *modified* from the standard one.

[43]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++ `regex`es, 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 - gamerank ** (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 (≈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: *musicalscales, 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 - romanholidays ** (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, drunkvigenere, 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 $_{16}C_2 \times _{14}C_2 \times \ldots \times _2C_2 = \frac{16!}{2!^8} \approx 8 \times 10^{10}$ 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:

```java
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!
  }
}
```

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

```python
from datetime import date                     # Python code for task 3
s = date(2010, 8, 9)                          # CP1 launch date
t = date.today()
print(s.strftime("%a"))                       # 'Mon', %A for 'Monday'
print("{} day(s) ago".format((t-s).days))     # ans grows over time
```

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

```cpp
#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);
}
```

```cpp
#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
}
```

```cpp
#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));
}
```

```cpp
#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");
  }
}
```

```java
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));
  }
}
```

```java
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][0-9]\\b+", "***"));
  }
}
```

```java
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[44] `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 `strncmp(line, ".......", 7) == 0`. Note that `strncmp(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.

---

[44]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 `String`s.

(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://zoj.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 ≈ 258 written exercises and the ≈ 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	**Getting Started**	155	$\approx 36\%$	$\approx 4.5\%$
1.6	**The Ad Hoc Problems**	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[1] 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

---

[1]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*[2] 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|ml`

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]`

---

[2]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`[3].

This data structure is similar to the static array, except that it is designed to handle runtime resizing natively[4]. 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 `iterator`s for traversing the contents of `vector`s. You can also directly do lexicographical comparison of the values in two `vector`s 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`

---

[3]Python `array` is not really needed as Python `list` is simpler to use.
[4]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[5]/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'[6]—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).

---

[5]We refer to the randomized version of Quick Sort that has $O(n \log n)$ time complexity in expectation.
[6]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`
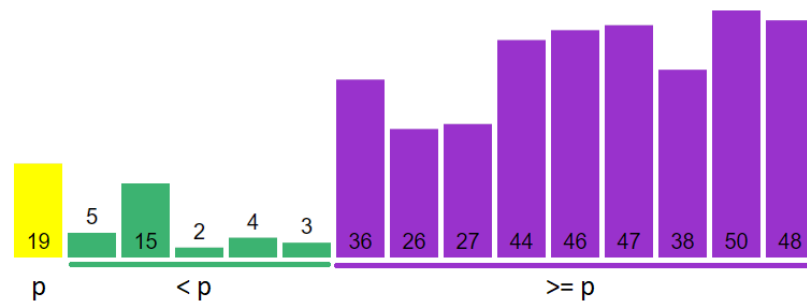
Figure 2.1: Sorting Visualization, Example of First Partition of Quick Sort

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 ↑, string last_name ↓ (descending order), string first_name ↑).

2*. N fractions ($\frac{numerator}{denominator}$) in ↑ (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.
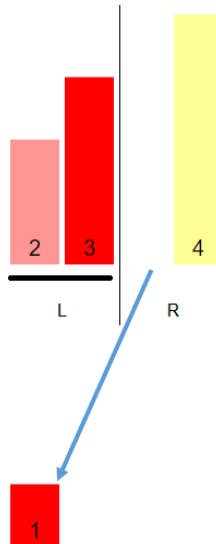
Figure 2.2: Sorting Visualization, Example of the Merge Operation of Merge Sort

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]` $\forall i \in$ `[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
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[7]. 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[8] 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
   ```

---

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

[8]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 != 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[9] the bitwise AND operation `S &= ~(1<<j)`.

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

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
```

[9]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 `LSOne(S)`[10].

```
 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 = 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 $\mathtt{mask} = (18)_{10} = (10010)_2$, 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[11] built-in bit manipulation functions[12]: `__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`

---

[10]This `LSOne(S)` operation is quite versatile and is used several times in this book.
[11]Java has `Integer` class that has these functionalities too, e.g., `bitCount`, `numberOfTrailingZeros`.
[12]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., $S = (40)_{10} = (101\underline{1}000)_2 \rightarrow S = (32)_{10} = (10\underline{0}000)_2$.

4. Turn on the last zero in S, e.g., $S = (41)_{10} = (1010\underline{0}1)_2 \rightarrow S = (43)_{10} = (1010\underline{1}1)_2$.

5. Turn off the last consecutive run of ones in S
   e.g., $S = (39)_{10} = (100\underline{111})_2 \rightarrow S = (32)_{10} = (100\underline{000})_2$.

6. Turn on the last consecutive run of zeroes in S
   e.g., $S = (36)_{10} = (1001\underline{00})_2 \rightarrow S = (39)_{10} = (1001\underline{11})_2$.

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

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

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`[13], Java `long`[14], or OCaml `Int64`.

One way to implement Big Integer library is to store the Big Integer as a (long) string[15]. 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:

```
num1         =   1,000,000,000,000,000,000,000
num2         =                             173
                ------------------------------ +
num1 + num2  =   1,000,000,000,000,000,000,173
```

We can also compute `num1` * `num2` as:

```
num1         =   1,000,000,000,000,000,000,000
num2         =                             173
                ------------------------------ *
                 3,000,000,000,000,000,000,000
                70,000,000,000,000,000,000,00
               100,000,000,000,000,000,000,0
                ------------------------------ +
num1 * num2 = 173,000,000,000,000,000,000,000
```

Addition and subtraction are the two 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[16]. 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[17] have such a feature and thus it is a good idea to use Python or Java to deal with these Big Integer problems.

---

[13]GCC has a 128-bit integer type `__int128` but it won't help here.

[14]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.

[15]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.

[16]Good news for IOI contestants. IOI tasks usually do not require contestants to deal with Big Integer.

[17]Pure C++ users must built 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).

```python
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 Big Integer routine in C++). In our code, we use the fast Java I/O: BufferedReader and PrintWriter instead of Scanner and System.out.println.

```java
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");                            // divide to F friends
    }
    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`[18].
Java `LinkedList`.
Python `list`.
OCaml `List` module.

Although this data structure almost always appears in data structure and algorithm text-books, 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[19] 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 ≈ 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[20]).
Python `list`.
OCaml `Queue` module.

---

[18]This `forward_list` library is very rarely used as its space saving feature is not usually needed.
[19]OCaml does not have built-in resizeable array.
[20]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[21]).
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|ml`

Exercise 2.2.5.1*: We can also use a *resizeable* array (C++ STL `vector`/Java `ArrayList`) to implement an efficient[22] 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[23] 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[24] queue using *two* resizeable arrays?

---

[21]The Java `Deque` is also an *interface* that is usually instantiated with Java `LinkedList`.
[22]Where all operations remain $O(1)$.
[23]Where all operations remain $O(1)$.
[24]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 1 2 9 3</u> / * + 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>5</u> - *	4 7 5	An operand
4 1 2 9 3 / * + 5 <u>-</u> *	4 7 5	Take 5 and 7, compute 7 - 5, push 2
4 1 2 9 3 / * + 5 - <u>*</u>	4 2	Take 2 and 4, compute 4 * 2, push 8
4 1 2 9 3 / * + 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[25]) 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
4 * (1 + 2 * (9 / 3) - 5)		4	Immediately output
4 * (1 + 2 * (9 / 3) - 5)	*	4	Put to stack
4 * (1 + 2 * (9 / 3) - 5)	* (	4	Put to stack
4 * (1 + 2 * (9 / 3) - 5)	* (	4 1	Immediately output
4 * (1 + 2 * (9 / 3) - 5)	* (+	4 1	Put to stack
4 * (1 + 2 * (9 / 3) - 5)	* (+	4 1 2	Immediately output
4 * (1 + 2 * (9 / 3) - 5)	* (+ *	4 1 2	Put to stack
4 * (1 + 2 * (9 / 3) - 5)	* (+ * (	4 1 2	Put to stack
4 * (1 + 2 * (9 / 3) - 5)	* (+ * (	4 1 2 9	Immediately output
4 * (1 + 2 * (9 / 3) - 5)	* (+ * (/	4 1 2 9	Put to stack
4 * (1 + 2 * (9 / 3) - 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 $O(n)$?

---

[25]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, thisaintyourgrandpascheckerboard.*

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[26]

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.*

---

[26]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 - throwns* * (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 `stack`s 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 `queue`s)
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*[27] 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 ourself), 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`[28] 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 `struct`s 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*[29] 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.
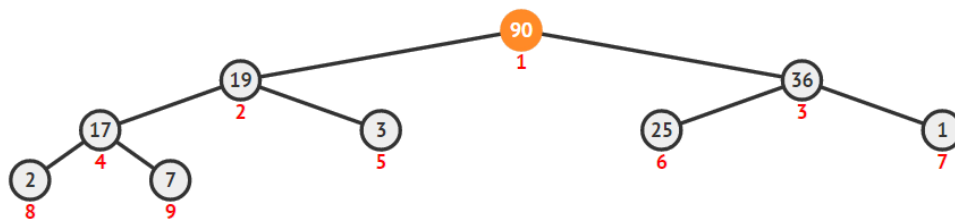
Figure 2.4: Example of a Binary (Max) Heap with Max Item (90) Highlighted

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

---

[27]The contents of a dynamic data structure is frequently modified via insert/delete/update operations.
[28]We can also use the faster C++ STL `unordered_map` with $O(1)$ performance if the keys do not have to be ordered.
[29]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[30] $O(\log n)$ time. The implementation[31] 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[32] is by processing the items one by one and creating a Max[33] 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[34]. 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`).

---

[30]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.

[31]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.

[32]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 $O(n + k \log n)$.

[33]The default `partial_sort` produces the smallest k items in ascending order.

[34]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 Visu-Algo, 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|ml`

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[35] 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[36] 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).
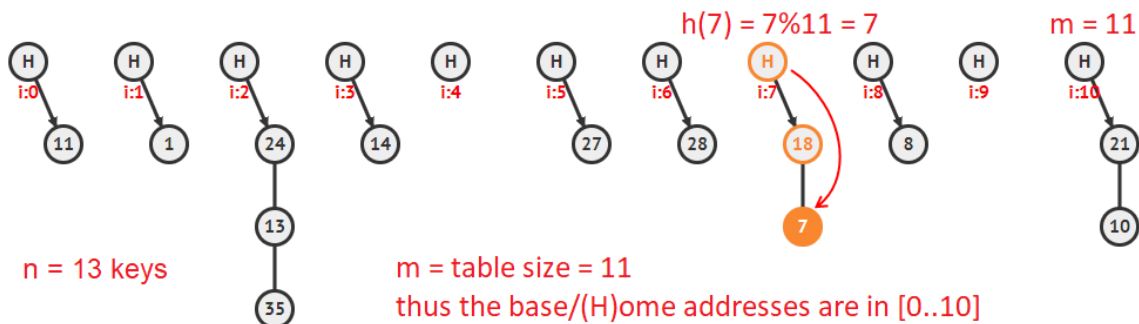
Figure 2.5: Search of Key 7 in a Hash Table with $m = 11$ and using Separate Chaining

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[37] data pairs whereas the C++ STL `unordered_set` only stores the keys. We

---

[35]Note that questions about hashing frequently appear in interviews for IT jobs.

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

[37]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[38]), `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[39] $O(1)$ 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[40] $2 \times M$ 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' → 3, 'W' → 10, ..., 'I' → 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.

[38]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.

[39]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.

[40]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 $\rightarrow$ value pairs in a *sorted array* of `struct`s 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.
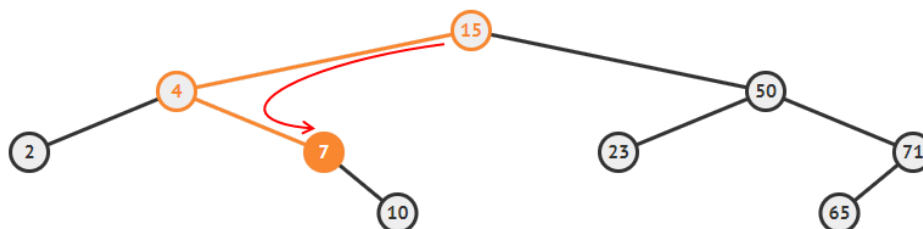
Figure 2.6: Example of Searching a Key (7) in a balanced BST (bBST)

Balanced Binary Search Tree (bBST) and Its Library Solutions

Implementing *bug-free* balanced BSTs such as the Adelson-Velskii Landis (AVL)[41] or Red-Black (RB)[42] 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[43] `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[44]. By mastering these two C++ STL libraries (or Java APIs), we can save a lot of precious coding time during contests!

---

[41]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.

[42]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.

[43]If there are duplicate elements, we may want to use the C++ STL `multimap` or `multiset` instead.

[44]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[45]. 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[46] 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.

---

[45]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.

[46]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 $O(\log n + (n\text{-}1) \times \log n) = O(n \log n)$? 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[47] 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., `Select(k) = v`, then the ranking of v is k, i.e., `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.

---

[47]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 $A[0..q\text{-}1]$.

3. $q+1 < k$, the desired answer is inside the right partition, e.g., in $A[q\text{+}1..n\text{-}1]$.

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.

```cpp
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|ml`

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[48] `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 $O(n + \frac{n}{2} + \frac{n}{4} + ... + \frac{n}{n}) = O(n \times (\frac{1}{1} + \frac{1}{2} + \frac{1}{4} + ... + \frac{1}{n})) \leq O(2n) = O(n)$.

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.

---

[48]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 augment 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:

```cpp
#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[49] (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 - alphabetspam* * (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 1 000 bus numbers; use 1D Boolean array)
7. *Kattis - freefood* * (only 365 days in a year)

   Extra UVa: *00755*.

   Extra Kattis: *floppy*, *hardware*, *relocation*.

---

[49]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*, *iwannabe*, *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: *baconeggsandspam, cakeymccakeface, fantasydraft, hardwoodspecies, 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[50].

- V=7, E=8 • Tree? **No** • Complete? **No** • Bipartite? **No** • DAG? **No** • Cross? **No**

Adjacency Matrix

	0	1	2	3	4	5	6
0	0	1	1	0	0	0	0
1	1	0	1	1	0	0	0
2	1	1	0	0	1	0	0
3	0	1	0	0	1	0	0
4	0	0	1	1	0	1	0
5	0	0	0	0	1	0	1
6	0	0	0	0	0	1	0

Adjacency List

0:	1	2	
1:	0	2	3
2:	0	1	4
3:	1	4	
4:	2	3	5
5:	4	6	
6:	5		

Edge List

0:	0	1
1:	0	2
2:	1	2
3:	1	3
4:	2	4
5:	3	4
6:	4	5
7:	5	6
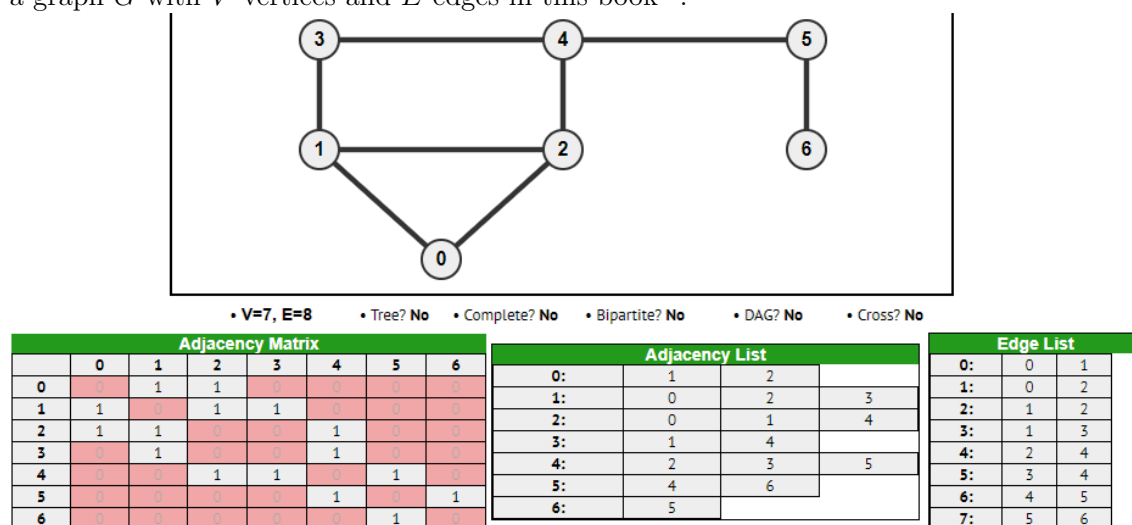
Figure 2.7: Graph Data Structure Visualization, Undirected/Unweighted Graph

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 `list`s 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*[51] 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[52]. For a weighted graph, set `AM[u][v] = weight(u, v)` if there is an edge between vertex u-v with `weight(u, v)` and zero otherwise. (Standard)

---

[50]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.

[51]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.

[52]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[53] 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, ∀u ∈ [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 ≈ 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[54] 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[55], usually (`w`, `u`, `v`), i.e., weight `w` plus the two endpoints `u` and `v`.
Using OCaml: `(int * int * int) list`.

---

[53]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.

[54]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.

[55]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[56] 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|ml`

Vertex Labels that are not $\in [0..V\text{-}1]$

So far, we assume that all vertices are labeled nicely, i.e., labeled with integer indices in a nice range of $[0..V\text{-}1]$. 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 $[0..V\text{-}1]$ (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

---

[56]`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 ($E = (V \times (V\text{-}1)/2) - L$; $L \leq 10K$). 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:
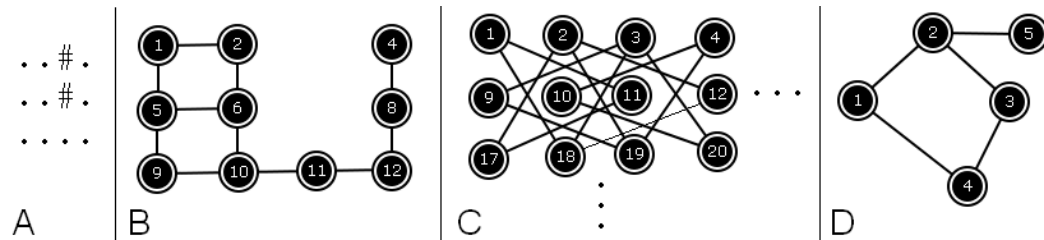
Figure 2.8: Implicit Graph Examples

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[57] and if both are '.' (see Figure 2.8—B).

2. The graph of chess knight movements on an 8×8 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.

---

[57]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 `set`s 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 `set`s! 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[58] 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 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

---

[58]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, `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.
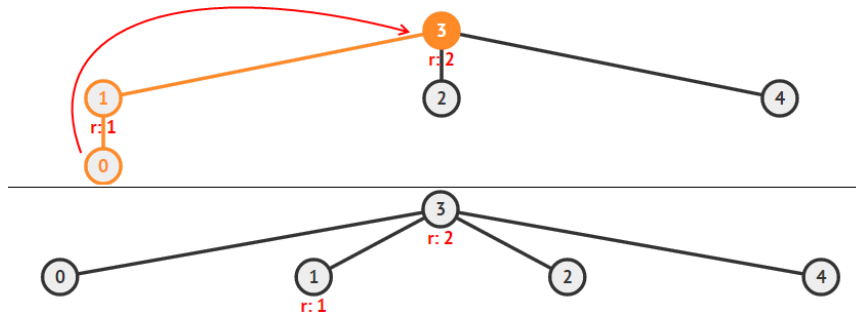
Figure 2.10: Top: `findSet(0)`, Bottom: The Subsequent Path Compression

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 = findSet(i)`, `y = findSet(j)`, and `x != 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 `rank[x]` $\leq$ `rank[y]`, otherwise we swap `x` and `y` first.

If `rank[x]` $<$ `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 `rank[x]` == `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 `rank[1]` to 1.

In Figure 2.11—middle, `unionSet(2, 3)` sets `p[2]` to 3 and `rank[3]` to 1.
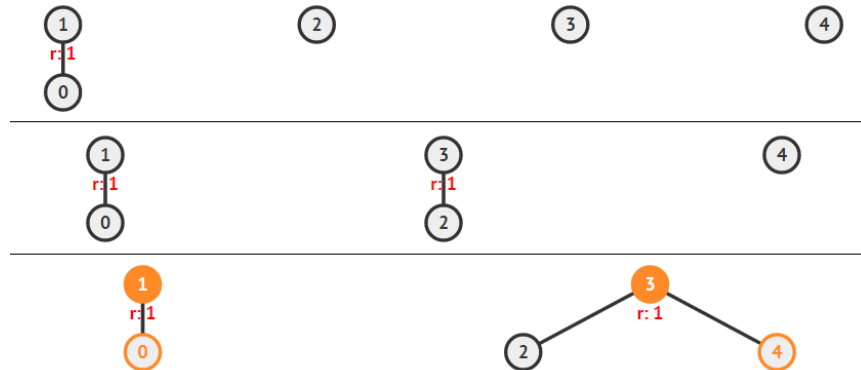
Figure 2.11: `unionSet(0, 1)` $\rightarrow$ `(2, 3)` $\rightarrow$ `(4, 3)` and `isSameSet(0, 4)`

In Figure 2.11—bottom, when we call `unionSet(4, 3)`, we get `rank[findSet(4)] = rank[4] = 0` which is smaller than `rank[findSet(3)] = rank[3] = 1`, so we set `p[4] = 3` *without* changing the height of the resulting tree (`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 `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 `rank[1] = 1`) and `p[3] = y = 3` (and `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 `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[59] in the representative item of the combined set.

```cpp
#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;
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
  }
};
```

---

[59]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|ml`

Exercise 2.4.2.1: Given N disjoint sets: $\{0, 1, 2, \ldots, N\text{-}1\}$, 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: $\{0, 1, 2, \ldots, N\text{-}1\}$, please create a sequence of `unionSet(i, j)` operations to create a tree with rank $= \log_2(N)$. Is it possible to create a tree with rank $> \log_2(N)$? Note that the 'union by rank' heuristic is used.

Exercise 2.4.2.3: Given N disjoint sets: $\{0, 1, 2, \ldots, N\text{-}1\}$, 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[60] 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, ..., i`.

Index/ Score	Frequency f	Cumulative 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[61] as it stores `RSQ(1, i)` $\forall i \in$ `[1..m]` where `m` is the largest integer index/score[62]. In the example above, we have `m = 10`, `RSQ(1, 1) = 0`, `RSQ(1, 2) = 1`, ..., `RSQ(1, 6) = 7`, ..., `RSQ(1, 8) = 10`, ..., 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).

---

[60]The test scores do not have to be sorted.

[61]`RSQ(i, j)` of an array A is the sum of `A[i] + A[i+1] + ... + A[j]`.

[62]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, `LSOne(90) = LSOne((1 011 0$\underline{1}$0)$_2$) = ($\underline{1}$0)$_2$ = 2`.

The Fenwick Tree[63] 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*[64] keys. These integer keys fall within the fixed range `[1..m]`—skipping[65] 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-LSOne(i)+1)..i]` of the frequency array `f`, i.e., `ft[i]` stores the cumulative frequency of items `{i-LSOne(i)+1, i-LSOne(i)+2, i-LSOne(i)+3, .., 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-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[66]. In Figure 2.12, the bottom side shows the raw frequency array `f` for each index `i`.

Operation: $O(\log m)$ `rsq(j)`
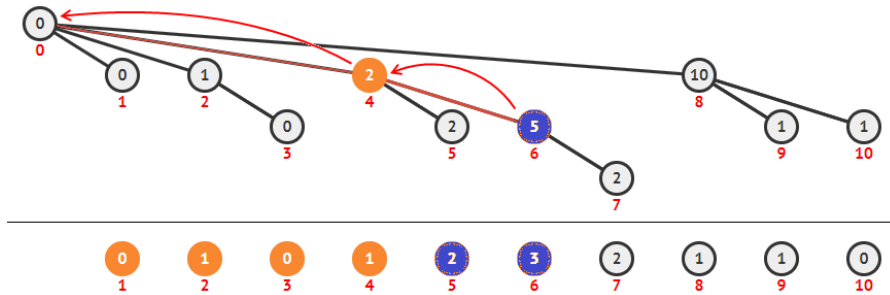
Figure 2.12: Example of `rsq(6)` on Fenwick (Interrogation/Query) Tree

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-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, `rsq(6) = ft[6]+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

---

[63]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.

[64]Recall that every (non-negative) integer has a unique binary representation.

[65]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`.

[66]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} = $(1\underline{1}0)_2$ 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 $1 \le i \le j \le m$ 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 `rsq(6)-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)`.
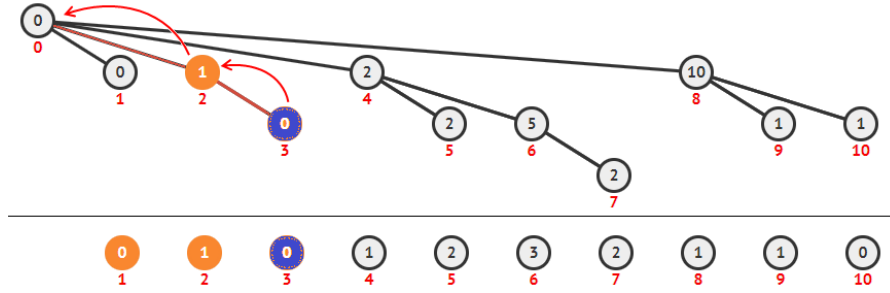
Figure 2.13: Example of `rsq(3)` on Fenwick (Interrogation/Query) Tree

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+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.
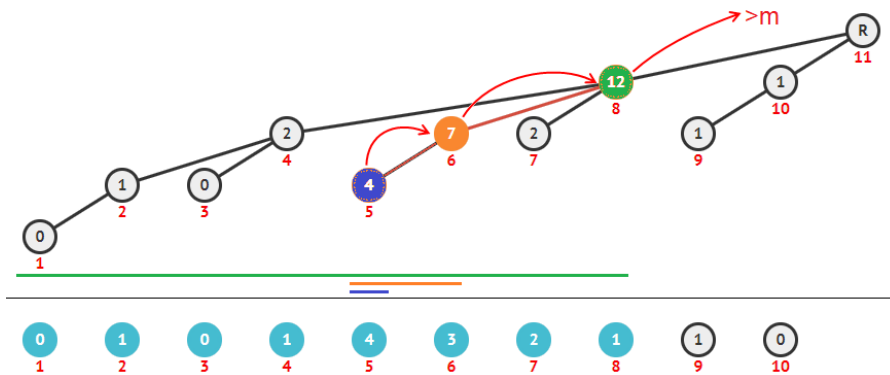
Figure 2.14: Example of `update(5, 2)` on Fenwick (Updating) Tree

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'` = $(101)_2$ + $(00\underline{1})_2$ = $(110)_2$ = 6_{10}, and `i''` = $(110)_2$ + $(0\underline{1}0)_2$ = $(1000)_2$ = 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 $0/i\text{-}1/m\text{-}1$ 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.

```cpp
#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 `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 $\forall \texttt{i} \in \texttt{[1..m]}$. 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).

```
          1111111111222222222233
    1234567890123456789012345678901

    ..............._................=0    (a)
    .............11111..............=1    (b)
    ...........11222 11.............=2    (c)
    ...1111....1122211.............=2    (d)
    ...11121111223 2211.............=3    (e)
```
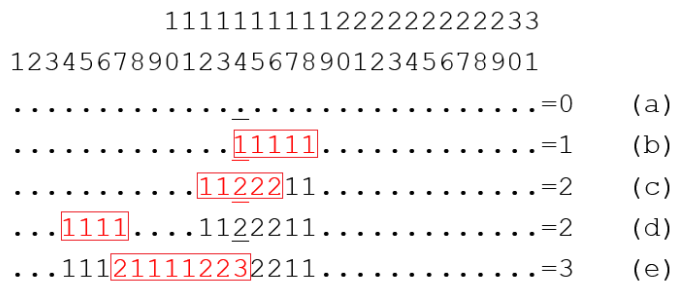
Figure 2.15: RUPQ Example; Point Queries $i = 14$ are Underlined

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 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 $[ui, ui + 1, \ldots, n]$ have +v value and `update(uj+1, -v)` makes all indices in $[uj + 1, uj + 2, \ldots, n]$ have -v value again, canceling the previous update. Therefore, `rsq(1, i)` for all $i \in [ui, ui + 1, \ldots, uj - 1, uj]$ will be correctly updated by +v.

```
            1111111111222222222233
            1234567890123456789012345678901
            ..............................=0    (a)
            ..............1....X...........=1    (b)
            ............1.1..X.X...........=2    (c)
            ...1...X...1.1..X.X...........=2    (d)
            ...1..1X...1.1X.X.X...........=3    (e)
```
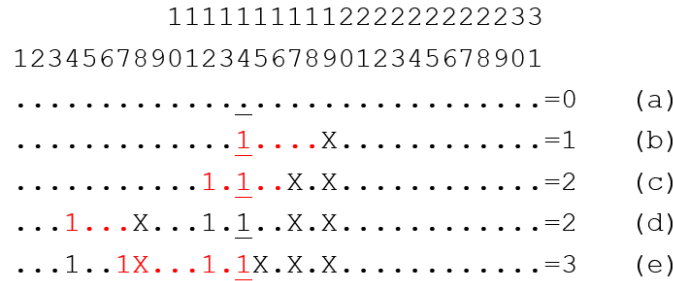
Figure 2.16: RUPQ Example; X denotes -1; Point Queries $i = 14$ are Underlined

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.

```
            1111111111222222222233
            1234567890123456789012345678901
            ..............................=0    (a)
            ..............11111...........=5    (b)
            ............1122211...........=4    (c)
            ...1111....1122211...........=4    (d)
            ...11121111223221 1...........=8    (e)
```
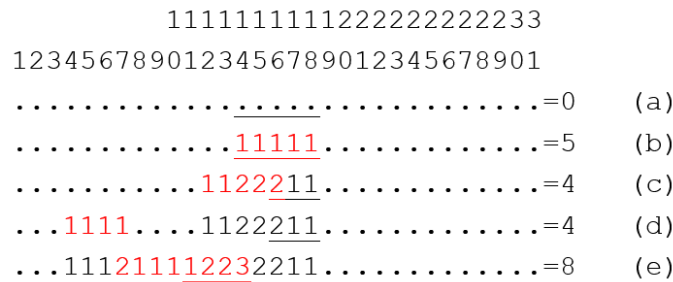
Figure 2.17: RURQ Example; Range Queries are Underlined

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 `rsq(i, j) = rsq(1, j) - 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, .., 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

.......=0     (a)
..111..=0     (b)
..111..=2     (c)
..111..=3     (d)
```

Figure 2.18: RURQ Explanation; Range Queries are <u>Underlined</u>

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 $ui \le j \le uj$, 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 $[1..ui\text{-}1]$ 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 $3 \le j \le 5$, 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 $[1..ui\text{-}1]$ and $[uj\text{+}1..j]$ are not updated.
   We already set `cancellation_factor = purq.update(ui, v*(ui-1))` earlier, but doing so we overdo the cancelation factor for $[uj\text{+}1..j]$.
   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.

```cpp
#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
  }                                               // [ui, ui+1, .., uj] +v
  ll point_query(int i) { return ft.rsq(i); }     // rsq(i) is sufficient
};

class RURQ  {                                     // RURQ variant
private:                                          // needs two helper FTs
  RUPQ rupq;                                      // one RUPQ and
  FenwickTree purq;                               // 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|ml`

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., `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..j]. This is known as the Range Minimum[67] Query (RMQ) problem[68].

For example, given an array A of size $n = 7$ below, 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, RMQ(3, 4) = 15, RMQ(0, 0) = 18, RMQ(0, 1) = 17, RMQ(4, 6) = 11, and 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] = ∞ (shown here as 99) that will not change the 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 != 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.

---

[67]The opposite Range *Maximum* Query problem is identical to this Range Minimum Query problem.

[68]Segment Tree can also be used to answer dynamic Range *Sum* Query (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 $O(1 + 2 + 4 + 8 + \ldots + 2^{\log_2 n}) = O(2 \times n)$ (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 $p = 1$).
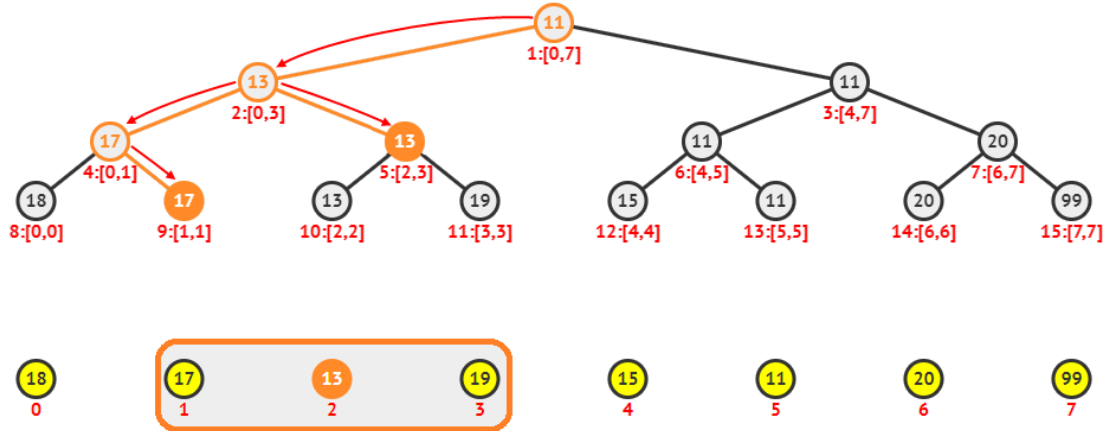
Figure 2.19: Segment Tree of `A` = $\{18, 17, 13, 19, 15, 11, 20, \infty\}$ and `RMQ(1, 3) = 13`

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[69] segment than the desired range in `RMQ(1, 3)`. From

---

[69]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[70] 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[71] 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.
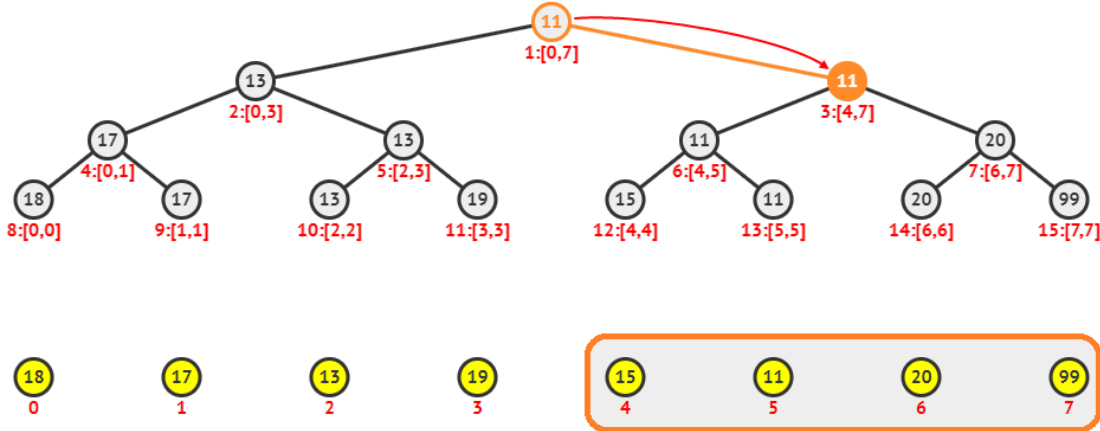
Figure 2.20: Segment Tree of A = {18, 17, 13, 19, 15, 11, 20, ∞} and RMQ(4, 7) = 11

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.

---

[70]Segment p:[L,R] is said to be outside query range [i,j] if i > j.
[71]Segment p:[L,R] is said to be inside query range [i,j] if (L >= i) && (R <= 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.
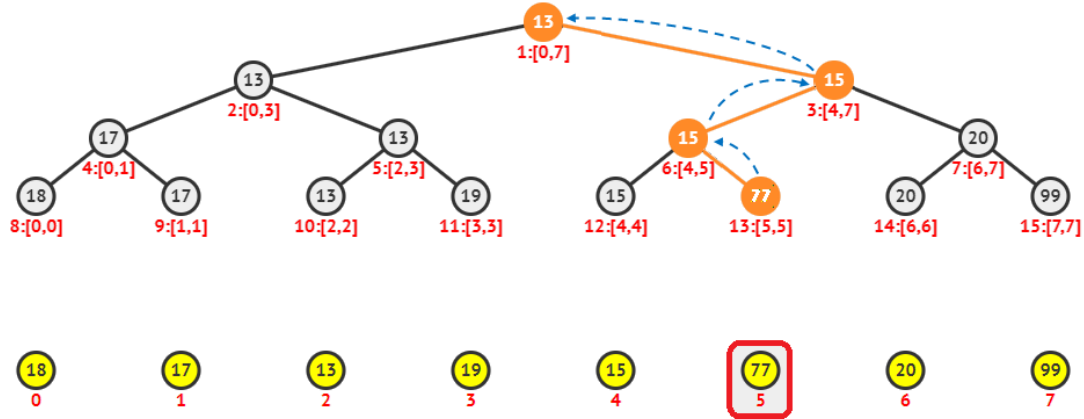
Figure 2.21: Updating `A` to {18, 17, 13, 19, 15, **77**, 20, ∞}

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) → `6:[4,5]` (`st[6]` = 15 as `min(15, 77)` = 15 now) → `3:[4,7]` (`st[3]` = 15 as `min(15, 20)` = 15 now) → `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` in 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**[72] **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]` → `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 → `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]` → `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.

---

[72]This Lazy Technique appears several times in this book and is a worthwhile technique to be studied.
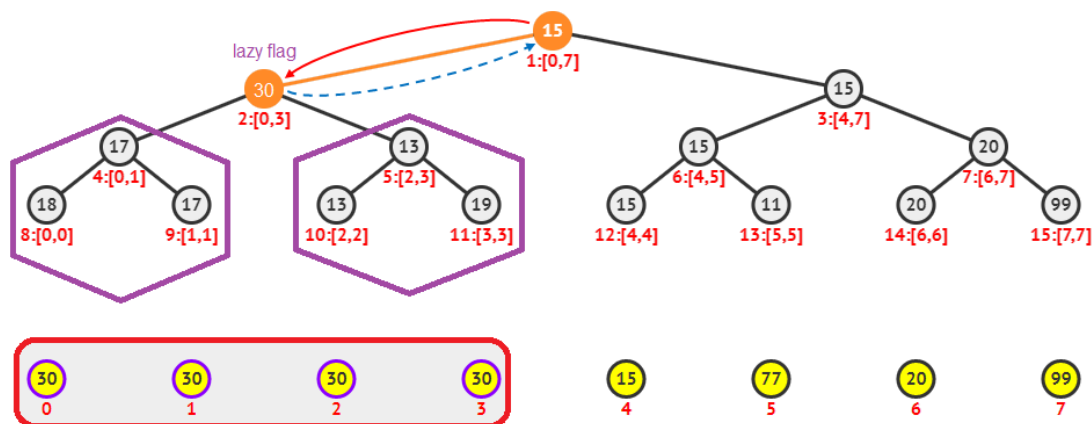
Figure 2.22: Updating A to {**30, 30, 30, 30**, 15, 77, 20, ∞}

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 at affected paths. For this example, see the paths in Figure 2.23.

The path is: `1:[0,7]` → `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 → `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]` → `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.
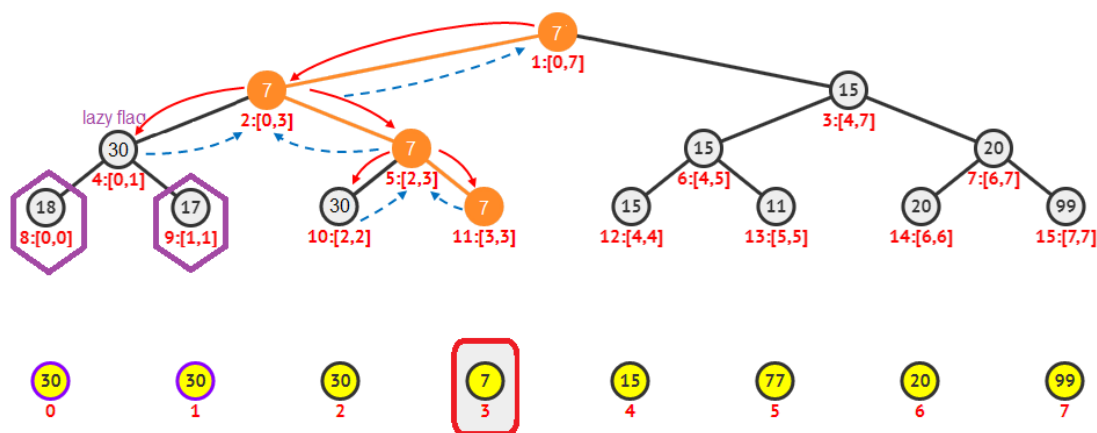
Figure 2.23: Updating A to {30, 30, 30, **7**, 15, 77, 20, ∞}

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];                                 // [L..R] has same value
      if (L != R)                                      // not a leaf
        lazy[l(p)] = lazy[r(p)] = lazy[p];             // propagate downwards
      else                                             // 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        ));
  }
```

```cpp
  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|ml`

Exercise 2.4.4.1: Using a similar Segment Tree as in the Exercise above, answer the queries `RSQ(1, 7)` and `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 `RMQ(1, 7)` and `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 - abinitio* * (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) → 50 → 71 (found)
   `search(7)`: root (15) → 4 → 7 (found)
   `search(22)`: root (15) → 50 → 23 → 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 `Iterator`s). 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 $\to$ report the number of rows; EL $\to$ count the number of distinct vertices in all edges. To count the number of edges of a graph: AM $\to$ sum the number of non-zero entries in every row; AL $\to$ sum the length of all the lists; EL $\to$ 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)` $\forall i \in [1..N\text{-}1]$. 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)` $\forall i \in [0..N\text{-}1]$ to compress the paths from all `i`s 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 Tree**s (which are similar to Segment Trees) and **Quad Tree**s (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	**Linear DS**	230	≈ 56%	≈ 6.7%
2.3	**Non-Linear DS**	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 kth 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 kth 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 $\{0, 1, 2, \ldots, N\text{-}1\}$ 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[1] 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.

---

[1]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, `abcde/fghij = N`, where each letter represents a different digit. The first digit of one of the numbers is allowed to be zero, e.g., for N = 62, we have 79546/01283 = 62; 94736/01528 = 62.

Quick analysis shows that `fghij` can only range from `01234` to `98765` which is at most $\approx 100K$ possibilities. An even better bound for `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 `fghij`[2], we can get `abcde` from `fghij`×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 `abcdefghij` and tests if the first five digits `abcde` divided by the last five digites `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[3] 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`

---

[2]Notice that it is better to iterate through `fghij` and not through `abcde` in order to avoid the division operator so that we only work with precise integers. If we iterate through `abcde` instead, we may encounter a non-integer result when we compute `fghij = abcde/N`.

[3]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 $[-22 \ldots 22]$. 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|ml`

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`[4] 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|ml`

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[5]: 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

---

[4]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.

[5]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 += l[j];                        // if yes, process 'j'
  if (sum == X) break;                    // the answer is found
}
```

Note: The implementation above can be speed up about a factor of two[6] 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 = two_pow_j, get j
    sum += l[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`

---

[6]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 $1, 2, \ldots, n$, 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, $\ldots$, 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 8×8 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.
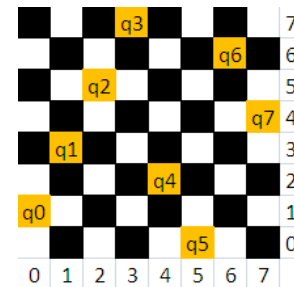
Figure 3.1: 8-Queens

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 `abs(i-k) == 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)))
      return false;                           // infeasible
  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 ($3 \leq n \leq \mathbf{15}$) 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] = true` and `rd[r+c] = 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.
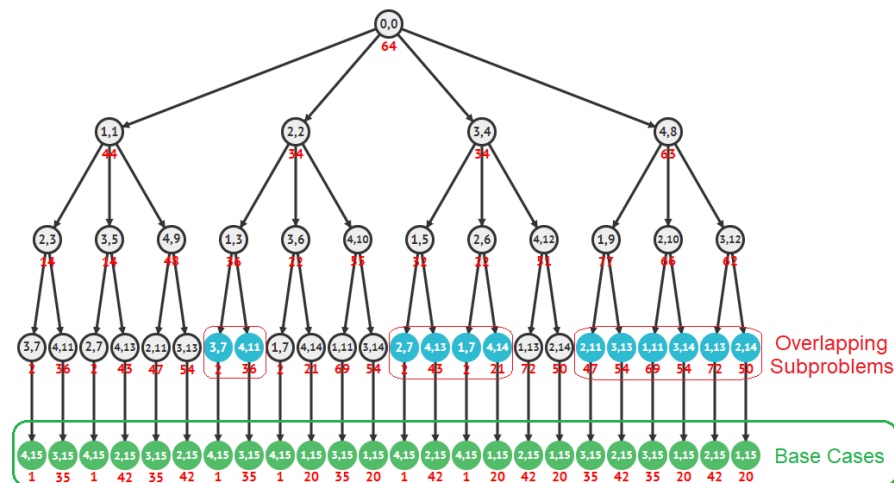
Figure 3.2: Recursion Tree of TSP with $n = 5$, also see Figure 4.42

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 $3 \le N \le 15$? What if $3 \le N \le 1000$? What if $3 \le N \le 100\,000$?

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 ≈ 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'[7] 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 $_{64}C_8$ 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.

---

[7]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 1025×1025) 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[8]!

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 ($n = 20\,000, d = 50$), ≈ 51 times faster than the frontal attack! This is an AC solution.

● Gas bomb explosion location
⬭ Diffusion area, example with d = 1
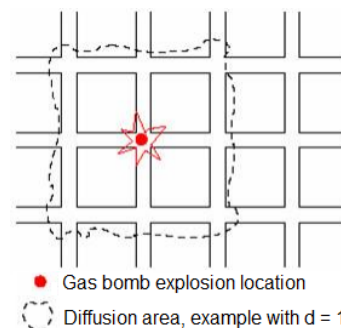
Figure 3.3: UVa 10360 [44]

---

[8]Although year 2020 CPU can compute $\approx 100M$ operations in ≈ 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[9] 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[10]: 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
```

---

[9]Most techniques mentioned in this tip are not good for general Software Engineering.
[10]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 `vector`s. Also, use 32-bit `int`s instead of 64-bit `long long`s 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 `Vector`s and `StringBuffer`s 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).

142

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[11] 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).

---

[11]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; $1 \le N \le 50$)
5. *Kattis - cardtrick2 ** ($n \le 13$, 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*, *towering*, *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[12], 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 missing_people $= 2 * a^2$, 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.*

---

[12]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 - dobra* * (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*[13], 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 ≥ 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 ≥ 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

---

[13]This is actually a (Min) Heap property albeit not on Binary Tree, see Section 2.3.1.
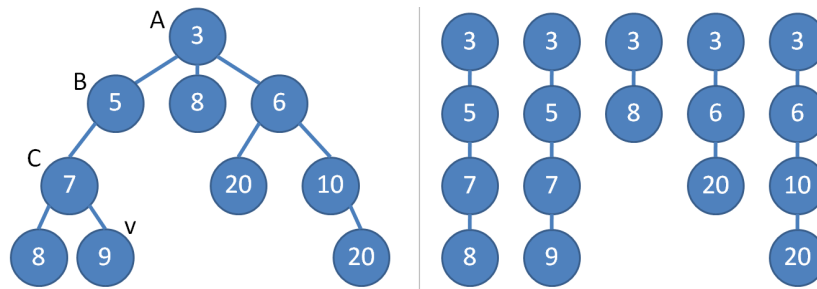
Figure 3.4: My Ancestor (all 5 root-to-leaf paths are sorted)

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**[14] 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 ≈ 0.

---

[14]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[15], 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[16] function (this is true for function $f(d)$ above).

a	b	$d = \frac{a+b}{2}$	status: $f(d, m, v, i)$	action
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 **few** iterations later . . .	. . .
. . .	. . .	576.190476	stop; error is now less than ϵ	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 ϵ 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 ≈ 40 iterations. Even if we use a smaller $\epsilon = $ `1e-15`, we will still only need ≈ 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.

---

[15]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.

[16]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).
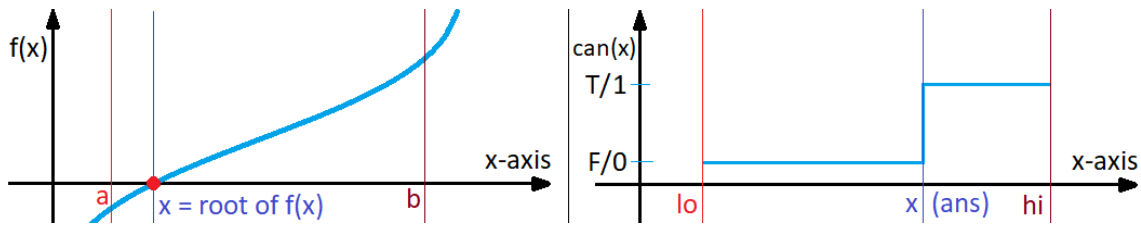
Figure 3.5: Monotone Function; Left: Bisection; Right: BSTA

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|ml`

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[17]. This unimodal function $f(x)$ in a range $[L..R]$ is formally defined as follows: $\forall a, b$ with $L \leq a < b \leq x$, 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.
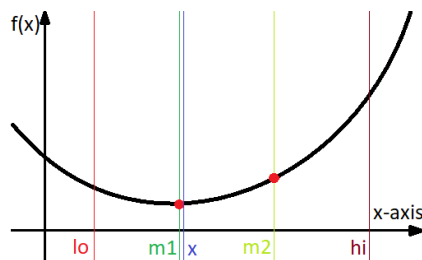
Figure 3.6: Ternary Search on a Unimodal Function

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[18].

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 ϵ and we can stop. This is efficient. The key part of Kattis - tricktreat code that uses ternary search is shown below.

---

[17]We can reverse the problem to find x such that $f(x)$ is maximum by reversing the signs of the constraints.
[18]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|ml`

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 - outofsorts* * (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*, *euclideantsp*, *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[19] 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[20], 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 → 17-$\underline{10}$ = 7 → 7-$\underline{5}$ = 2 → 2-$\underline{1}$ = 1 → 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.

---

[19]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.

[20]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[21].

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 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.
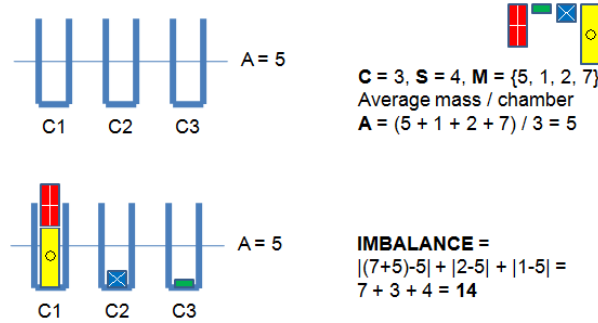
Figure 3.7: Visualization of UVa 00410 - Station Balance

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.
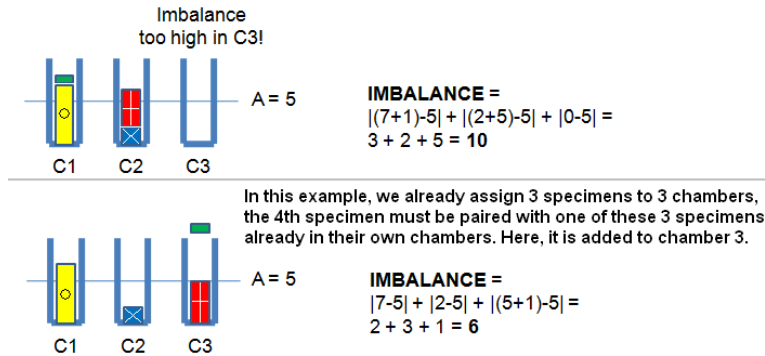
Figure 3.8: UVa 00410 - Observations

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 $M_1 \leq M_2 \leq \ldots \leq M_{2C-1} \leq M_{2C}$. In this example, $M = \{5, 1, 2, 7, 0, 0\} \rightarrow \{0, 0, 1, 2, 5, 7\}$. By adding dummy specimens and then sorting them, a greedy strategy 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 . . .

---

[21]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[22]! See Figure 3.9.
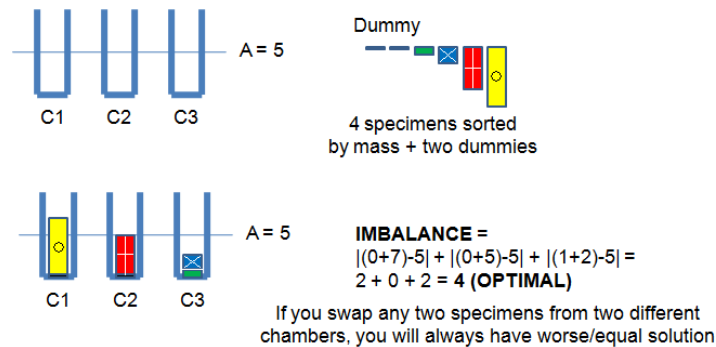
Figure 3.9: UVa 00410 - Greedy Solution

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}.
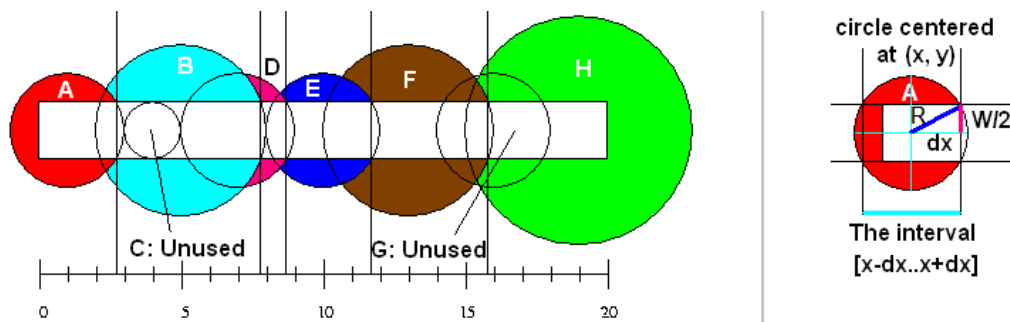
Figure 3.10: Kattis - grass/UVa 10382 - Watering Grass

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.

---

[22]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..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 > l) 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 < l)) 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 ($1 \le n, m \le 20\,000$). 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** $\le$ **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** $\le$ **H** (see Figure 3.11—right side, after sorting).
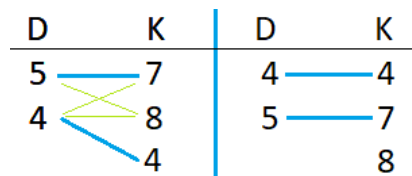
Figure 3.11: Kattis - loowater/UVa 11292 - The Dragon of ...

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 ($1 \le N \le 500K$) cities—each city must be assigned at least one box, the population size a_i of each city i ($1 \le a_i \le 5M$)—each person can only vote in his/her assigned box in his/her own city, and B ballot boxes ($N \le B \le 2M$), 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[23]. 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.

---

[23]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[24]:

- 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*.

---

[24]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 *state*s 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[25].

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[26] 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 <u>8</u> // the prices are not sorted in the input
Price of the 2 models of garment `g = 1` $\rightarrow$ 5 <u>10</u>
Price of the 4 models of garment `g = 2` $\rightarrow$ <u>1</u> 5 3 5

For this test case, the answer is 19, which *may* result from buying the <u>underlined</u> items (8+10+1). This is not unique, as solutions (6+10+3) and (4+10+5) are also optimal.

---

[25]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.

[26]But DP can also be the solution for a yes/no decision problem too.

However, suppose we have this test case B with $M = \mathbf{9}$ (**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[27]: $20 + 20 + \ldots + 20$ 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*[28] below (test case C):

Test case C with $M = 12$, $C = 3$:
3 models of garment `g = 0` $\rightarrow$ 6 <u>4</u> **8**
2 models of garment `g = 1` $\rightarrow$ <u>5</u> 10
4 models of garment `g = 2` $\rightarrow$ 1 5 <u>3</u> 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 <u>$4+5+3$</u> $= 12$, 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.

---

[27]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.

[28]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, $\forall$ `model` $\in$ `[1..k]` of current garment `g`,
`dp(g, b) = max(dp(g+1, b-price[g][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 $20 \times 20 \times \ldots \times 20$ 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[29].
    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-price*, where *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

---

[29]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[30] 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[31].

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[32] 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[33].

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
```

---

[30]Also see Book 2 for alternative data structures that can be used for this purpose.

[31]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`.

[32]Basic means "without fancy optimizations that we will see later in this section and in Book 2".

[33]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 sub-problem 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 20×201. 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).
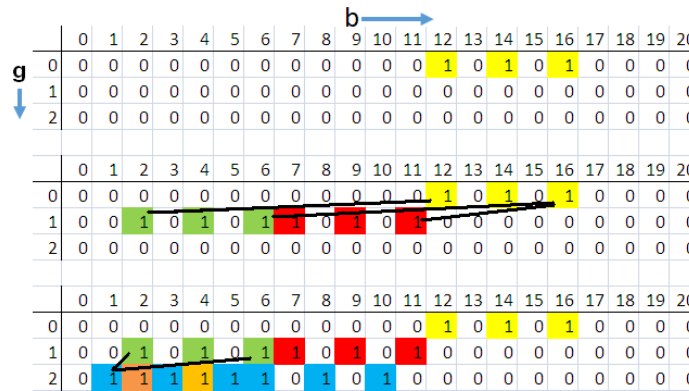
Figure 3.12: Bottom-Up DP (columns 21 to 200 are not shown for brevity)

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.

```cpp
// 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[34] 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 $\underline{20} \times 201$, we can simply store `reachable[2][b]` of size $\underline{2} \times 201$. 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|ml`

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

---

[34]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
Pros:	Pros:
1. It is a natural transformation from the normal Complete Search recursion	1. Faster if many sub-problems are revisited as there is no overhead from recursive calls
2. Computes the sub-problems only when necessary (sometimes this is faster)	2. Can save memory space with the 'space saving' technique
Cons:	Cons:
1. Slower if many sub-problems are revisited due to function call overhead (this is not usually penalized in programming contests)	1. For programmers who are inclined to recursion, this style may not be intuitive
2. If there are M states, an $O(M)$ table size is required, which can lead to MLE for some harder problems (except if we use alternative data structures shown in Book 2)	2. If there are M states, bottom-up DP visits and fills the value of *all* these M states even if many of the states are not necessary

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→12) = price 8, then (12→2) = price 10, then (2→1) = 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→16) = price 4, then (16→6) = price 10, then (6→1) = 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 → 6 4 8
Price of the 2 models of garment g = 1 → 10 6
Price of the 4 models of garment g = 2 → 7 3 1 5

Exercise 3.5.1.2: Is the following state formulation dp(g, 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] + +...+ 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 `RSQ(i, j)` in $O(1)$: `RSQ(0, j) = A[j]` and `RSQ(i, j) = A[j] - A[i-1]` $\forall i > 0$ using inclusion-exclusion principle. With this[35], 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
  }                                             // if it ever dips below 0
  printf("Max 1D Range Sum = %d\n", ans);       // should be 9
```

Source code: `ch3/dp/Max1DRangeSum.cpp|java|py|ml`

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 4×4 matrix ($n = 4$) in Table 3.3.A below has a 3×2 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.

---

[35]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.

Table 3.3: UVa 00108 - Maximum Sum

```
int maxSubRect = -127*100*100;                    // the lowest possible val
for (int i = 0; i < n; ++i)
  for (int j = 0; j < n; ++j)                      // start coordinate
    for (int k = i; k < n; ++k)
      for (int l = j; l < n; ++l) {                // end coord
        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
  }                                               // inclusion-exclusion
int maxSubRect = -127*100*100;                    // the lowest possible val
for (int i = 0; i < n; ++i)
  for (int j = 0; j < n; ++j)                      // start coordinate
    for (int k = i; k < n; ++k)
      for (int l = j; l < n; ++l) {                // end 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
      }
```

175

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]`,..., `A[n-1]`}, determine its Longest Increasing Subsequence (LIS)[36]. Note that these 'subsequences' are not necessarily contiguous. Example: $n = 8$, $A = \{\underline{-7}, 10, 9, \underline{2}, \underline{3}, \underline{8a}, 8b, 1\}$. The length-4 LIS is {-7, 2, 3, 8a} (it can also end with the second copy of 8, i.e., 8b).

---

[36]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 naïve 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 `LIS(i)` be the LIS ending at index `i`. We know that `LIS(0) = 1` as the first number in `A` is itself a subsequence. For `i ≥ 1`, `LIS(i)` is slightly more complex. We need to find the index `j` such that `j < i` and `A[j] < A[i]` and `LIS(j)` is the largest. Once we have found this index `j`, we know that `LIS(i) = 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\n", LIS(n-1));     // with O(n^2) DP
```

The answer is the largest value of `LIS(k)`, $\forall k \in$ `[0..n-1]`. However, if we use a sentinel value `A[n] = 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 `LIS(n)-1`.

There are clearly many overlapping sub-problems in LIS problem because to compute `LIS(i)`, we need to compute `LIS(j)` $\forall j \in$ `[0..i-1]`. However, there are only n distinct states, 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 `LIS(k)`. For example, `LIS(5)` is the optimal final state. In Figure 3.13, we can trace the arrows as follows: `LIS(5)` → `LIS(4)` → `LIS(3)` → `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[37] (where k is the length of the LIS) by maintaining an array that is *always sorted* and therefore amenable to binary search.

Let `vi L` and `vi L_id` be resizeable 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 `pos`, the position where we can either greedily lower the content of `L[pos]` to a lower number to facilitate potentially longer increasing subsequence in the future, or to extend the LIS by +1 if `pos == k`.

Note that the content of `L` is *not* the actual LIS. To facilitate reconstruction of the actual LIS (if asked[38]), we need to also remember the predecessor/parent array `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|ml`

---

[37]We classify this $O(n \log k)$ LIS algorithm ("patience sorting") under DP category due to legacy reason.
[38]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: $n = 11$, $A = \{\underline{-7}, 10, 9, 2a, 3a, 8a, 8b, \underline{1, 2b, 3b, 4}\}$ (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, `10`}`.

- For `A[2]` = 9, we replace `L[1]` so that we have a 'better' length-2 LIS ending:
  `L = {-7, `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, `2a`}`.

- We insert `A[4]` = 3a at `L[2]` so that we have a longer LIS, `L = {-7, 2a, `3a`}`.

- We insert `A[5]` = 8a at `L[3]` so that we have a longer LIS, `L = {-7, 2a, 3a, `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, `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, `2b`, 8a}`.

- For `A[9]` = 3b, we improve `L[3]` so that `L = {-7, 1, 2b, `3b`}`.

- We insert `A[10]` = 4 at `L[4]` so that we have a longer LIS, `L = {-7, 1, 2b, 3b, `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[39]: 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[40] ignore or take a particular item (hence the term 0-1 for ignore/take). Assume that $1 \le n \le 1000; 1 \le S \le 10\,000$.

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.

---

[39]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?

[40]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[41] 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 $1 \le n \le 1000; 1 \le V \le 10\,000$ (also see Section 3.4.1).

Example 1: $V = 10$, $n = 2$, `coinValue` $= \{1, 5\}$; We can use:
A. Ten 1 cent coins $= 10 \times 1 = 10$; Total coins used $= 10$
B. One 5 cents coin + Five 1 cent coins $= 1 \times 5 + 5 \times 1 = 10$; Total coins used $= 6$
C. Two 5 cents coins $= 2 \times 5 = 10$; Total coins used $= 2 \to$ 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: $V = 7$, $n = 4$, `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 $4+3$)!

---

[41]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, 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 + change(1-1)`, as `1 + change(1-5)` is infeasible (returns ∞).
`change(2)` = 2, from `1 + change(2-1)`, as `1 + change(2-5)` is also infeasible (returns ∞).
... same thing for `change(3)` and `change(4)`.
`change(5)` = 1, from `1 + change(5-5)` = 1 coin, smaller than `1 + change(5-1)` = 5 coins.
... and so on until `change(10)`.
The answer is in `change(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[42] 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
}
```

---

[42]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[43] 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|ml`

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[44] 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.

	A	B	C	D
A	0	20	42	35
B	20	0	30	34
C	42	30	0	12
D	35	34	12	0
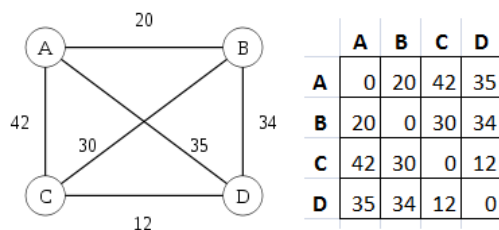
Figure 3.15: A Complete Weighted Graph K_4

$O(n!)$ 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 $A-B-C-best$ *sequence of* $(n-3)$ *other cities that finally returns to A* clearly overlaps the tour $A-C-B-the$ *same best sequence of* $(n-3)$ *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

---

[43]If V is large such that $nV > 1M$, this DP solution is not feasible even with the space saving technique!

[44]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`$= 18_{10} = 10010_2$ implies that items (cities) $\{2, 5\}$ have *not* been visited[45] yet. We will use fast bit manipulation techniques like `LSOne(S)` to quickly identify which '1' bit(s) are still available[46].

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[47] $n \approx [18..19]$ 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|ml`

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.

---

[45]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.

[46]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.

[47]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: `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 `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 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: $0 + 20, 1 + 19, 2 + 18, 3 + 17, \ldots, 20 + 0$.

Mathematically, the number of ways can be expressed as $^{(n+k-1)}C_{(k-1)}$ (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)
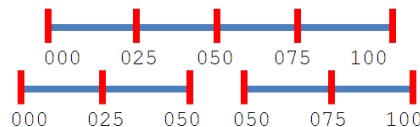
Figure 3.16: Cutting Sticks Illustration

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[48] 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, the original `A`, 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.

---

[48]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(LIS(i)+LDS(i)-1), $\forall i \in [0 \ldots n\text{-}1]$; 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 lexico-graphically 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: $(k+1)^n$; 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	**Complete Search**	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	**Dynamic Programming**	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 ≈ 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[1] (polynomial) solutions. Some do not have them yet[2] (see Book 2). In this relatively big chapter with lots of figures, we discuss graph problems[3] 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[4], 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

---

[1]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.

[2]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.

[3]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.

[4]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[5]) 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[6]. 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).

---

[5]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.

[6]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`.
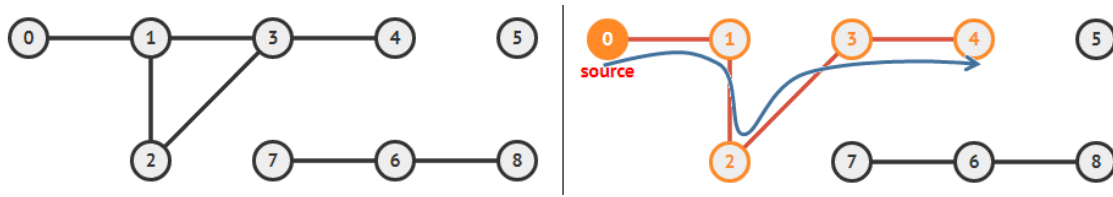
Figure 4.1: Left: Sample Graph, Right: Running `dfs(0)` on Sample Graph

Note that this sequence of visitation depends very much on the way we order the neighbors of a vertex[7], i.e., the sequence $0 \to 1 \to 3 \to 4$ (backtrack to 3) $\to 2$ 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);
  }
}
```

---

[7] 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).

```cpp
// 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
    }
  }
```
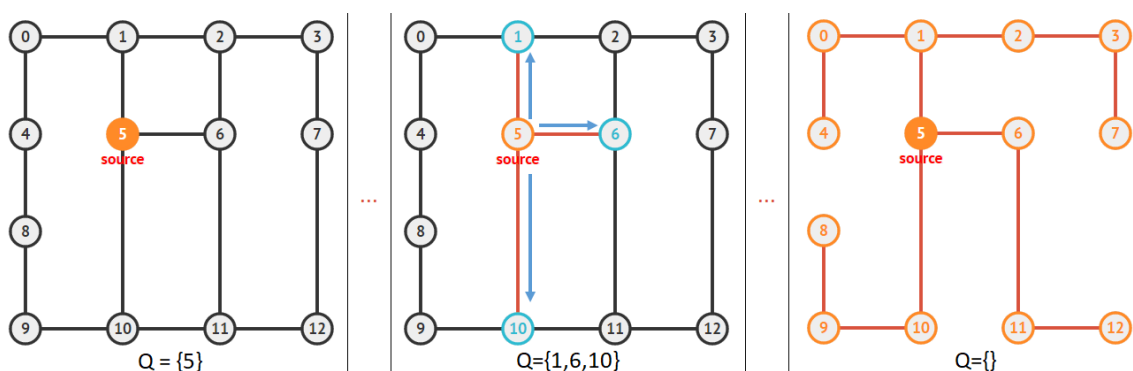

Figure 4.2: Example Partial Animation of BFS, see VisuAlgo for a Live Animation

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|ml`

Exercise 4.2.4.1: What are the minimum and maximum number of CCs in an undirected graph G with V vertices and E $(0 \le E \le V \times (V-1)/2)$ 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* $(1 \le V, E \le 200\,000; 1 \le K \le V)$. 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
// LLLLLLLLL          LLLLLLLLL
// LLWWLLWLL          LL..LLWLL                  // The size of CC
// LWWLLLLLL (R2,C1)  L..LLLLLL                  // with one 'W'
// LWWWLWWLL          L...L..LL                  // at (R2, C1) is 12
// LLLWWWLLL ======>  LLL...LLL
// LLLLLLLLL          LLLLLLLLL                  // Notice that all these
// LLLWWLLWL          LLLWWLLWL                  // connected 'W's are
// LLWLWLLLL          LLWLWLLLL                  // replaced with '.'s
// LLLLLLLLL          LLLLLLLLL                  // 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.
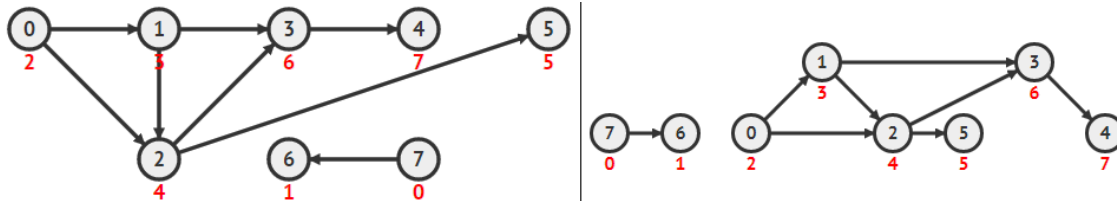
Figure 4.3: Left: A DAG, Right: The Same DAG Redrawn in its Topological Sort Order

```cpp
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
}
```

```cpp
// 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`

200

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.

```cpp
// 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|ml`

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[8]) 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|ml`

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!

---

[8]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*[9], respectively. With the help of one more vertex state: EXPLORED (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: EXPLORED 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: EXPLORED to another vertex v with state: EXPLORED 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: EXPLORED to another vertex v with state: VISITED.
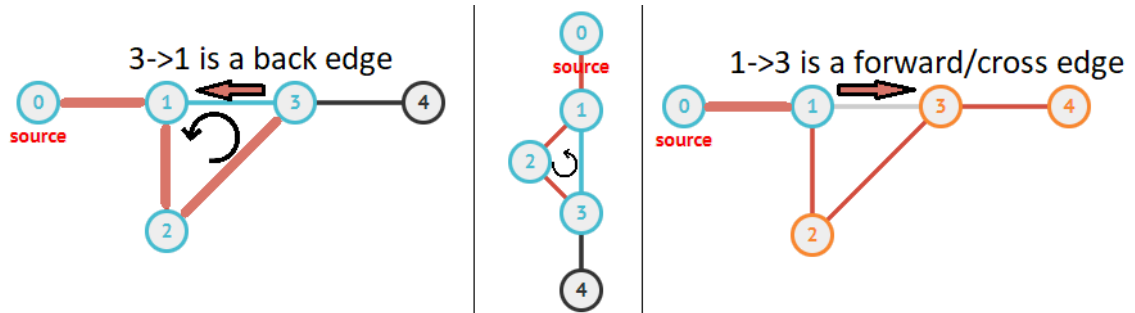
Figure 4.4: dfs(0) when Run on the First CC of Sample Graph in Figure 4.1

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 $0 \to \underline{1 \to 2 \to 3 \to 1}$ is a (true) cycle and we classify edge $(3 \to 1)$ as a back edge, whereas many other edges, e.g., $0 \to 1 \to 0$ is not a cycle but it is just a bi-directional edge (0-1).

---

[9]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:

```cpp
void cycleCheck(int u) {                              // check edge properties
  dfs_num[u] = EXPLORED;                              // color u as EXPLORED
  for (auto &[v, w] : AL[u]) {                        // C++17 style, w ignored
    printf("Edge (%d, %d) is a ", u, v);
    if (dfs_num[v] == UNVISITED) {                    // EXPLORED->UNVISITED
      printf("Tree Edge\n");
      dfs_parent[v] = u;                              // a tree edge u->v
      cycleCheck(v);
    }
    else if (dfs_num[v] == EXPLORED) {                // EXPLORED->EXPLORED
      if (v == dfs_parent[u])                         // differentiate them
        printf("Bidirectional Edge\n");               // a trivial cycle
      else
        printf("Back Edge (Cycle)\n");                // a non trivial cycle
    }
    else if (dfs_num[v] == VISITED)                   // EXPLORED->VISITED
      printf("Forward/Cross Edge\n");                 // rare application
  }
  dfs_num[u] = VISITED;                               // color u as VISITED/DONE
}
```

```cpp
// 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 `EXPLORED`/`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.
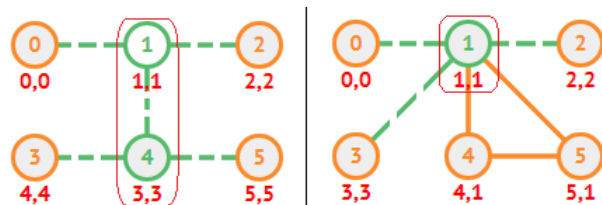
Figure 4.5: Two More DFS Attributes: `dfs_num` and `dfs_low`

Suppose for the graph in Figure 4.5—right side, the sequence of visitation is 0 (at iteration 0) → 1 (1) → 2 (2) (backtrack to 1) → 3 (3) (backtrack to 1) → 4 (4) → 5 (5). At this point in the DFS spanning tree, there is an important back edge that forms a cycle, i.e., edge 5 → 1 that is part of non-trivial cycle 1 → 4 → 5 → 1. This causes vertices 1 (itself), 4 (indirectly), and 5 (the vertex that discovers back edge 5 → 1) to all able to reach vertex 1 (with `dfs_num` 1). Thus `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 `dfs_low(v)` ≥ `dfs_num(u)`, then u is clearly an articulation vertex. This is because the fact that `dfs_low(v)` is *not smaller* than `dfs_num(u)` implies that there is *no back edge* from a vertex in the subtree rooted at v that can reach another vertex w with a lower `dfs_num(w)` than `dfs_num(u)`. A vertex w with lower `dfs_num(w)` than vertex u with `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).
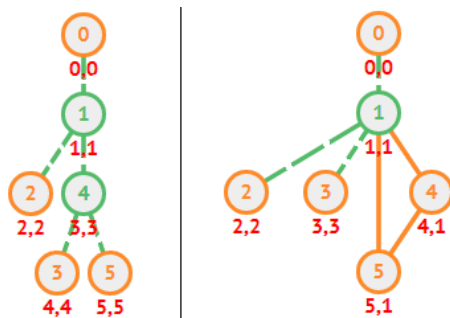
Figure 4.6: Finding Articulation Points with `dfs_num` and `dfs_low`

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 → 2, we see that `dfs_low(2)` ≥ `dfs_num(1)` (vertex 2 can only reach ancestor of vertex 1 via articulation point vertex 1) and similarly in edge 4 → 5, we also see that `dfs_low(5)` ≥ `dfs_num(4)`.

On the graph in Figure 4.6—right, only vertex 1 is the articulation point, because for example in edge 1 → 5, `dfs_low(5)` ≥ `dfs_num(1)`. On the other hand, vertex 4 is not an articulation point because when we examine edge 4 → 5, we see that `dfs_low(5)` < `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 → 1).

The process to find bridges is similar. When $\texttt{dfs\_low(v)} > \texttt{dfs\_num(u)}$, 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 $\texttt{dfs\_low(4)} \leq \texttt{dfs\_num(1)}$, 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 $\texttt{dfs\_low(4)} = 1$ (that other path is actually path $4 \to 5 \to 1$). 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[10] in Section 4.2.6). For example in Figure 4.7—left, the decreasing finishing order of the 8 vertices is $\{0, 1, 3, 4, 5, 7, 6, 2\}$. In turns out that on the transposed graph, these ordering can help us identify the "smallest SCC" (read [5] for the details).

First pass: Dec finish order {0,1,3,4,5,7,6,2} | Second pass on Transposed Graph
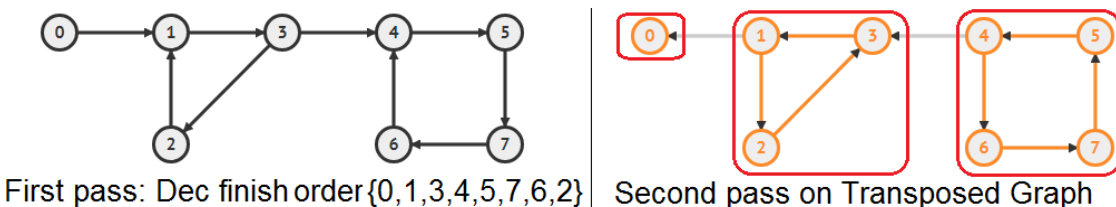
Figure 4.7: Execution of Two Passes Kosaraju's Algorithm

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).

---

[10]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.

```cpp
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
}
```

```cpp
// 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
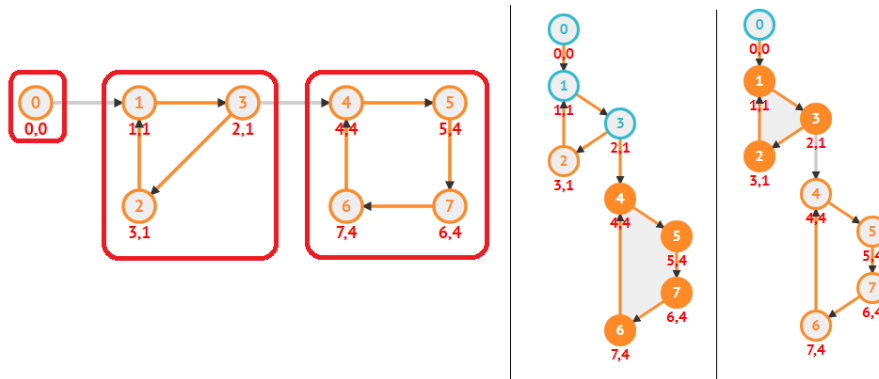
Figure 4.8: Left: Directed Graph; Middle+Right: DFS Spanning Tree Snapshots

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 $\{0, 1, 3, 2, \underline{4, 5, 7, 6}\}$ when vertex 4 is found as root of an SCC (`dfs_low(4)` = `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 $\{0, \underline{1, 3, 2}\}$ when vertex 1 is identified as the root of another SCC (`dfs_low(1)` = `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])                              // condition for update
      dfs_low[u] = min(dfs_low[u], dfs_low[v]);
  }
  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	*Usually* use less memory	Can solve SSSP
	Can find cut vertices, bridges, SCC	on unweighted graphs
Cons	Cannot solve SSSP	*Usually* use more memory
	on unweighted graphs	(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/dfsbfs`

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_map`s; 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!
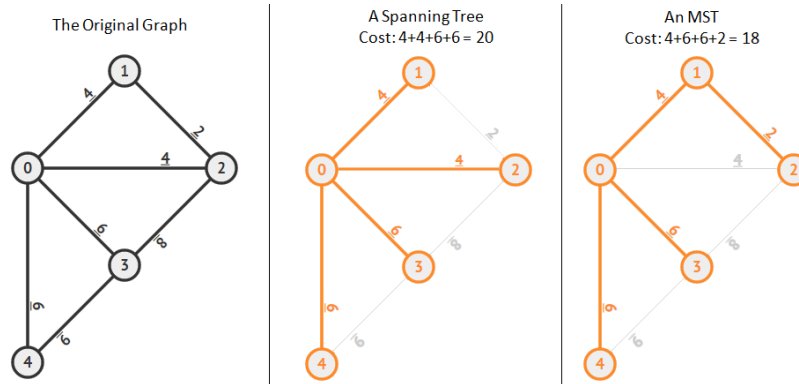
Figure 4.9: Example of an MST Problem

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[11] 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.

---

[11]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(sorting + trying to add each edge × 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.
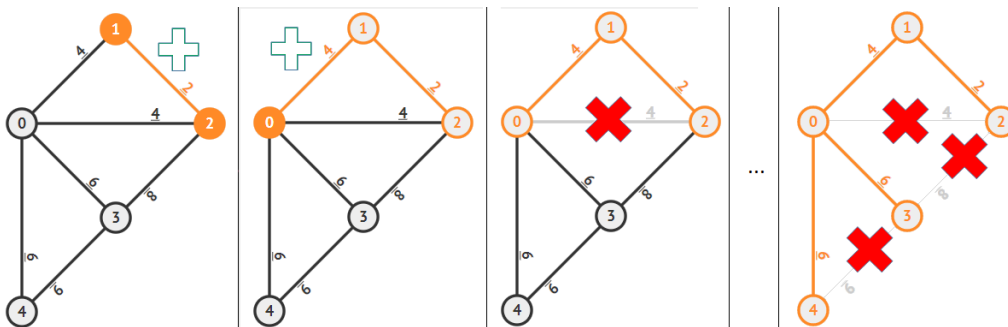
Figure 4.10: Animation of Kruskal's Algorithm for an MST Problem

---

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 Vojtêch 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(process each edge once × cost of enqueue/dequeue) $= O(E \times \log E) = O(E \log V)$.

```cpp
vector<vii> 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
```

```cpp
// inside int main() --- assume the graph is stored in AL, pq is empty
  int V, E; scanf("%d %d", &V, &E);
  AL.assign(V, vii());
  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);
```
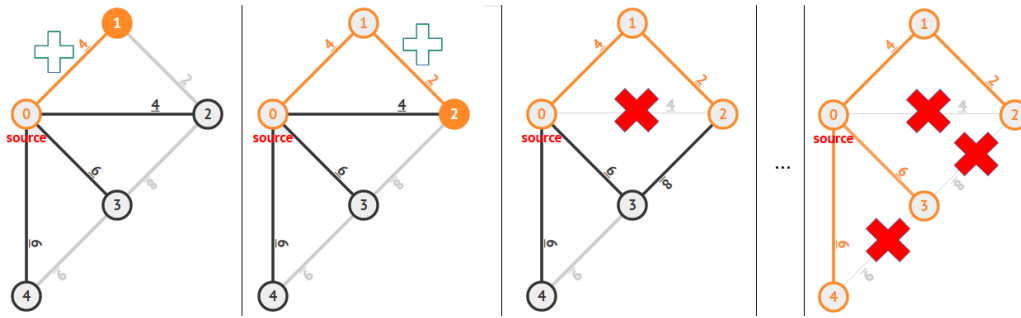
Figure 4.11: Animation of Prim's Algorithm for the same graph as in Figure 4.9—left

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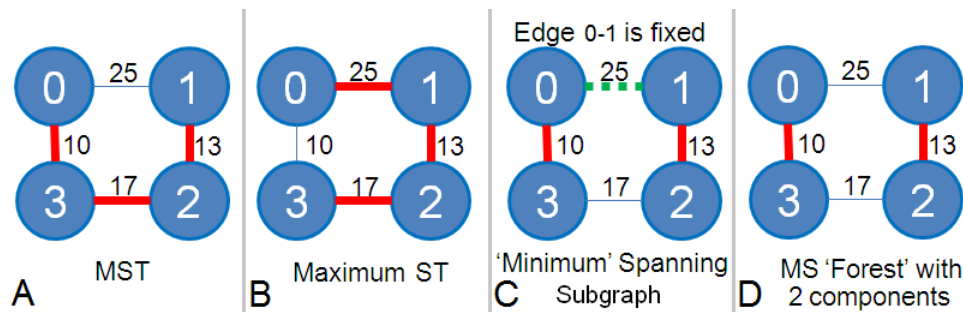
Figure 4.12: From left to right: MST, 'Maximum' ST, 'Minimum' SS, MS 'Forest'

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)
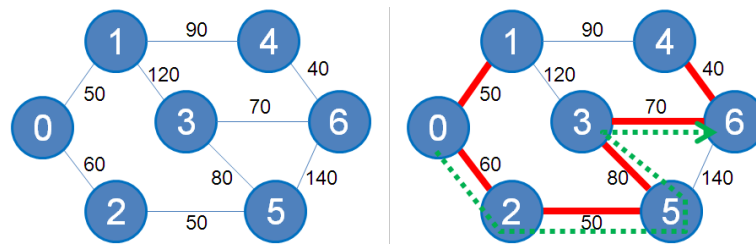
Figure 4.13: Minimax (UVa 10048 [44])

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 $E = V\text{-}1$ 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
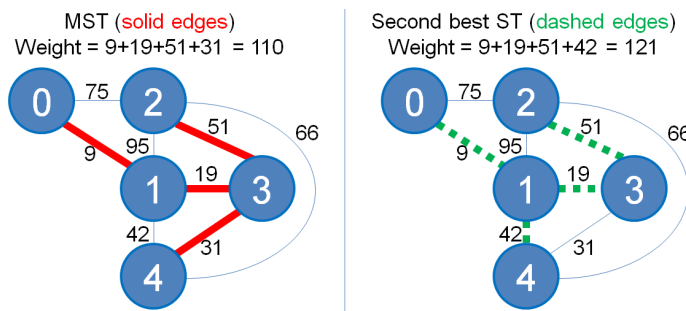
Figure 4.14: Second Best ST (from UVa 10600 [44])

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[12] edge is added into the MST. Here, edge 3-4 is taken out and edge 1-4 is added in.
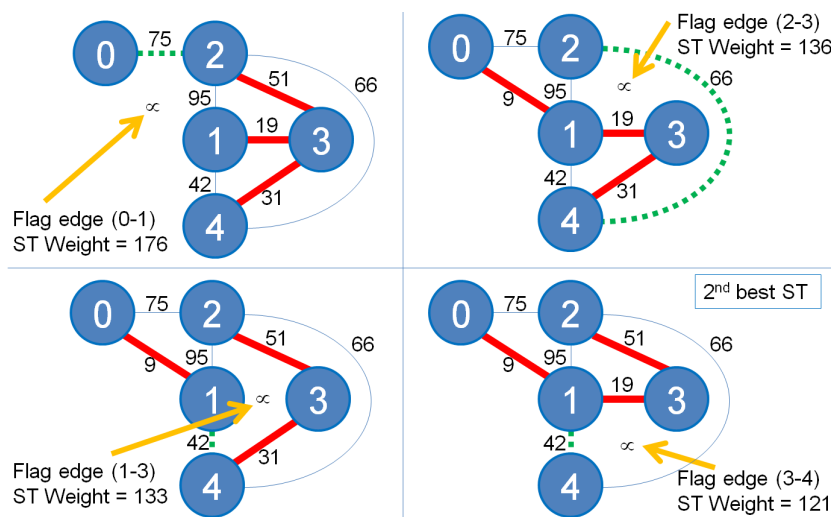
Figure 4.15: Finding the Second Best Spanning Tree from the MST

---

[12]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..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*, *jurrasicjigsaw*, *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*, *firetrucksarered*, *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 Vojtêch Jarník and rediscovered independently by Prim. Thus Prim's algorithm sometimes also known as Jarník-Prim algorithm.

Vojtêch Jarník (1897-1970) was a Czech mathematician. He developed the graph algorithm now known as Prim's algorithm. In the era of fast and widespread publication of scientific results nowadays, Prim's algorithm would have been credited to Jarník instead of Prim.

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 $s \to \ldots \to u \to \ldots \to v$!

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[13] 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

---

[13]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 `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 $5 \to 1 \to 2 \to 3 \to 7$. 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 `vi p` (see Section 2.4.1). Each vertex v remembers its parent u (`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+Deque

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
AAAAA...............CCCC.    // Solution: Walk 11 steps east from
AAAAA.AAA..........CCCC.     // the rightmost A to leftmost C in this row
AAAAAAAAA....###....CCCC.    // Then, from the rightmost C in this row,
AAAAAAAAA...............     // walk 2 steps south
AAAAAAAAA...............     // 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.

```cpp
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
}
```

```cpp
// 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]) {                 // C++17 style, w ignored
      if (dist[v] != INF) continue;              // already visited, skip
      dist[v] = dist[u]+1;
      p[v] = u;                                  // addition
      q.push(v);
    }
  }
  printPath(t), printf("\n");                     // addition
```

Source code: `ch4/sssp/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[14] 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 another 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 $3 \le n \le 4$ and $n \ge 5$.

2. Literally concentrate on corner cells and side cells.

3. What happen if the starting cell and the target cell are too close?

---

[14]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*[15] 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 $0 \to 3$ with weight 7 that is normally found by BFS, but a 'detour' path: $0 \to 1 \to 3$ 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 `dist[s] = 0` (the shortest path from s to s itself is clearly 0) while `dist[u] = ` ∞ for all other V-1 vertices that are not s. Dijkstra's algorithm uses a Priority Queue (`pq`) data structure of vertex information pair (`dist[u]`, u) to dynamically order (sort) the pairs by non-decreasing `dist[u]` values (vertex number u is unique). We insert V vertex information pairs of all V vertices into `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 `dist[u]` first (also see Section 3.4.1 about greedy algorithm with `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 `dist[s] = 0`) will be processed first while the rest (currently with unknown/infinitely large shortest path distance values) will be behind in `pq`. Then, Dijkstra's algorithm tries to relax each neighbor v of $u = s$. The `relax(u, v, w_u_v)` operation sets `dist[v] = min(dist[v], 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., `dist[v]`, will be lowered from initially ∞ into a (much) lower number. We also update (lower) that information in `pq` and let `pq` dynamically (re-)order the vertices based on non-decreasing `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 `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[16] C++ STL `set`/Java `TreeSet`/OCaml `Set`, we can update (lower) old (higher `dist[u]`, u) into new (lower `dist[u]`, u) by first deleting old (higher `dist[u]`, u) in $O(\log V)$ time and re-inserting new (lower `dist[u]`, u) also in $O(\log V)$ time.

Dijkstra's algorithm then repeats the same process until `pq` is empty: it greedily takes out vertex information pair (`dist[u]`, u) from the front of `pq` and relax each outgoing edge $u \to v$ of u, updating (lowering) `dist[v]` and the associated pair in `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 `pq` operations

---

[15]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.

[16]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 Dijkstra'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.

1. Figure 4.16—left: At the beginning, only `dist[s] = dist[0] = 0`, `set<ii> pq` initially contains $\{(0, 0), (\infty, 1), (\infty, 2), (\infty, 3), (\infty, 4)\}$.
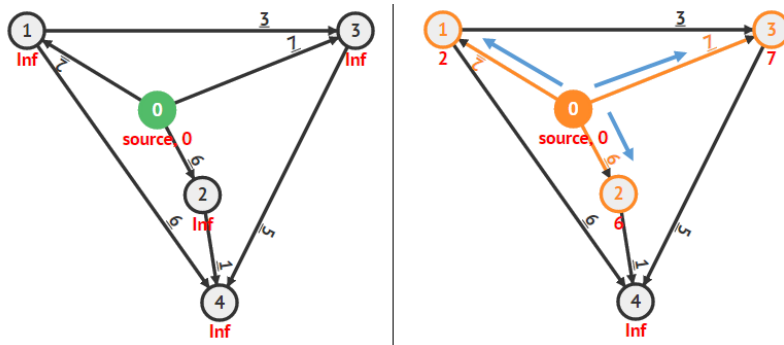
Figure 4.16: Dijkstra's Animation on a Weighted Graph (from UVa 00341 [44]), Steps 1+2

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 `dist[1] = 2`, `dist[2] = 6`, and `dist[3] = 7`. While doing this, we simultaneously update (lower) the keys in `set<ii> pq`. `set<ii> pq` now contains $\{(2, 1), (6, 2), (7, 3), (\infty, 4)\}$.

3. Figure 4.17—left: Dequeue the vertex information pair at the front of `pq`: $(2, 1)$. Relax edges incident to vertex 1 to get `dist[3] = min(dist[3], dist[1]+w(1,3))` `= min(7, 2+3) = 5` and `dist[4] = 8` and update the keys in `pq`. `set<ii> pq` now contains $\{(5, 3), (6, 2), (8, 4)\}$. By now, edge $0 \to 3$ is not going to be part of the SSSP spanning tree from $s = 0$.
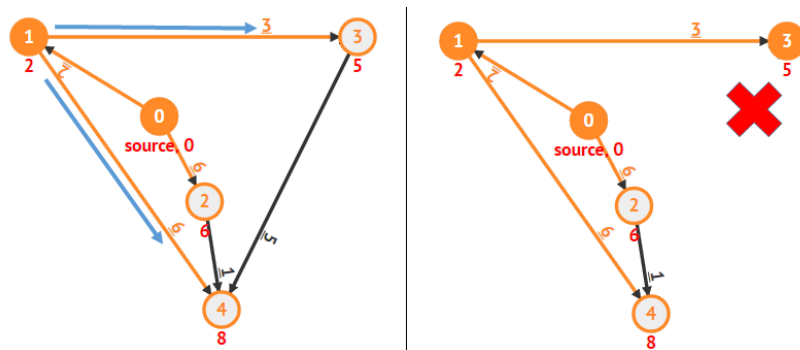
Figure 4.17: Dijkstra's Animation, Steps 3+4

4. Figure 4.17—right: We dequeue $(5, 3)$ and try to do `relax(3, 4, 5)`, i.e., $5+5 = 10$. But `dist[4] = 8` (from path $0 \to 1 \to 4$), so `dist[4]` is unchanged. `set<ii> pq` now contains $\{(6, 2), (8, 4)\}$. By now, edge $3 \to 4$ 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 `dist[4] = 7`. The shorter path from 0 to 4 is now $0 \to 2 \to 4$ instead of $0 \to 1 \to 4$. `set<ii> pq` now contains $\{(7, 4)\}$. By now, edge $1 \to 4$ is also not going to be part of the SSSP spanning tree from $s = 0$.
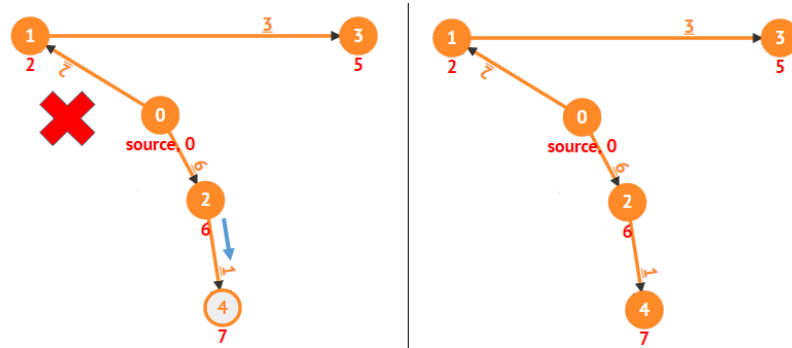
Figure 4.18: Dijkstra's Animation, Steps 5+6

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.

```cpp
// 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.

229

Modified Dijkstra's algorithm works 99% similar with the Original Dijkstra's algorithm as it also maintains a Priority Queue (`pq`) that stores the same vertex information pairs. But this time, `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 `pq` is empty: it greedily takes out vertex information pair (d, u) from the front of `pq`. If the shortest path distance from s to u recorded in d is greater than `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 $u \to v$, it will *always enqueue* a pair (newer/shorter distance from s to v, v) into `pq` and will *always leave the inferior pair* (older/longer distance from s to v, v) inside `pq`. This is called as 'Lazy Deletion' and it causes *more than one copy* of the same vertex in `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 `pq` update operations in Modified Dijkstra's only *lower* the `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 `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 `dist[s] = 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 `pq`: $(0, 0)$. Relax edges incident to vertex 0 to get `dist[1] = 2`, `dist[2] = 6`, and `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 `pq`: $(2, 1)$. Relax edges incident to vertex 1 to get `dist[3] = min(dist[3], dist[1]+w(1,3))` `= min(7, 2+3) = 5` and `dist[4] = 8` and immediately enqueue two more pairs in `pq`. `priority_queue<ii> pq` now contains $\{\underline{(5, 3)}, (6, 2), \underline{(7, 3)}, (8, 4)\}$.
   See that we have 2 entries of vertex 3 in `pq` with increasing distance from s. We do not immediately delete the inferior pair $(7, 3)$ from the `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 $0 \to 3$ 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 `dist[4] = 8` (from path $0 \to 1 \to 4$), so `dist[4]` is unchanged. `priority_queue<ii> pq` now contains $\{(6, 2), (7, 3), (8, 4)\}$.
   By now, edge $3 \to 4$ 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 `dist[4] = 7`. The shorter path from 0 to 4 is now $0 \to 2 \to 4$ instead of $0 \to 1 \to 4$. `priority_queue<ii> pq` now contains $\{(7, 3), \underline{(7, 4)}, \underline{(8, 4)}\}$ (2 entries of vertex 4). By now, edge $1 \to 4$ 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 > 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 pq for the standard $O(\log V)$ deletion of C++ STL `priority_queue` to work.
   `priority_queue<ii> pq` now contains only {(7, 4), (8, 4)}.
   We then dequeue (7, 4) and process it, but nothing changes.
   `priority_queue<ii> pq` now contains only {(8, 4)}.
   Finally, we dequeue (8, 4) but ignore it again as its `d > dist[4]` (i.e., 8 > 7).
   `priority_queue<ii> 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.

```cpp
// 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 $3 \to 4$.
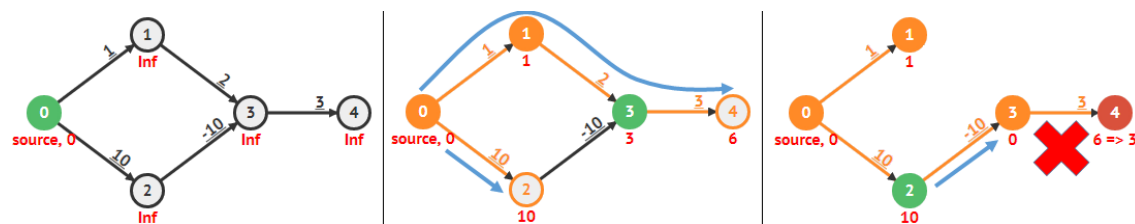
Figure 4.19: Original Dijkstra's Fails on a Negative Weight Graph, $s = 0$

In Figure 4.19—middle, we see that the Original Dijkstra's *wrongly* propagates shortest path distance from $0 \to 1 \to 3$ to vertex 4, causing vertex 4 to believe that the shortest path from source vertex 0 is $0 \to 1 \to 3 \to 4$ 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 $0 \to 2 \to 3$ 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 $0 \to 2 \to 3$ to vertex 4, causing vertex 4 to now have the correct shortest path of $0 \to 2 \to 3 \to 4$ of value 3.
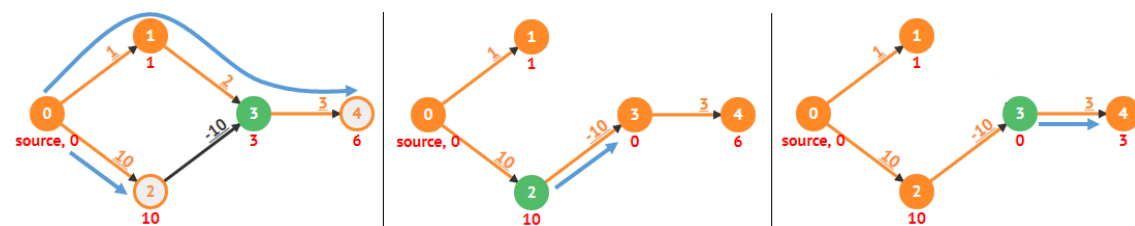
Figure 4.20: Modified Dijkstra's Can Work on a Non-Negative Cycle Graph, $s = 0$

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 $1 \to 2 \to 3 \to 1$ 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[17]. On such test case like in Figure 4.21, Modified Dijkstra's will first take the bottom path $0 \to 2 \to 4 \to 6 \to 8 \to 10$ with cost $0 + 0 + 0 + 0 + 0 = 0$ before finding $0 \to 2 \to 4 \to 6 \to 8 \to 9 \to 10$ 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 $0 \to 1 \to 2 \to 3 \to 4 \to 5 \to 6 \to 7 \to 8 \to 9 \to 10$ 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.
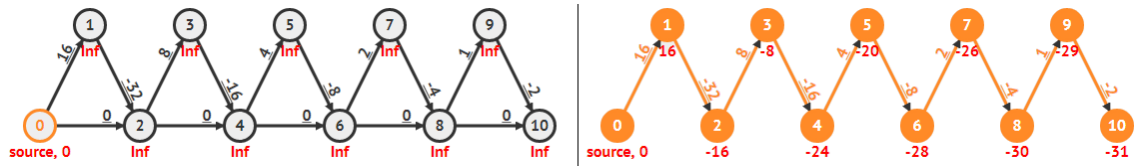
Figure 4.21: Modified Dijkstra's Can Be Made to Run in Exponential Time

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".

---

[17]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 `dist[s] = 0`, the base case. If we relax an edge (s, u), then `dist[u]` will have the correct value. If we then relax an edge (u, v), then `dist[v]` will also have the correct value. If we have relaxed all E edges V-1 times, then the shortest path from the source vertex to the furthest vertex from the source (which will be a simple path with V-1 edges) should have been correctly computed (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)                    // these two loops = O(E)
      if (dist[u] != INF)                          // important check
        for (auto &[v, w] : AL[u])                 // C++17 style
          dist[v] = min(dist[v], dist[u]+w);
```

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 $i+1, i+2, \ldots, i = V$-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.
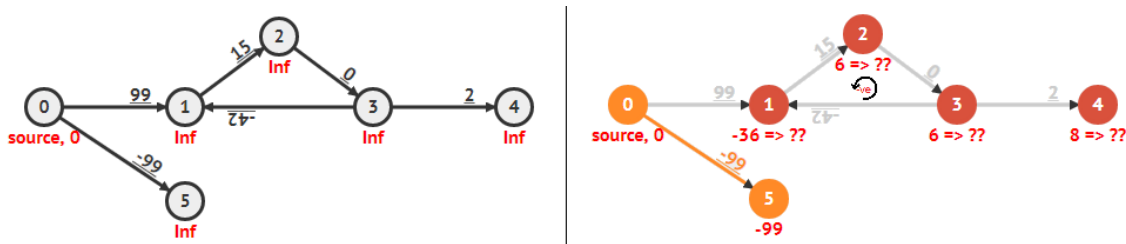
Figure 4.22: Bellman-Ford can detect the presence of negative cycle (UVa 00558 [44])

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, `dist[1]` = 72 and `dist[2]` = `dist[3]` = 114. After V-1 = 6−1 = 5 passes, `dist[1]` = -36 and `dist[2]` = `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 `dist[2]` = -36+15 = -21. This is lower than the current value of `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[18].

```
// 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])                      // C++17 style
        if (dist[v] > dist[u]+w)                      // should be false
          hasNegativeCycle = true;                    // 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`

---

[18]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[19]). 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:

```cpp
// 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[20]. 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!

---

[19]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.

[20]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[21] 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`

---

[21]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[22] 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**.

---

[22]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, onaveragetheyrepurple, 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 Dijstra'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: ∞ + negative weight is lower than ∞)
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 `dist[s][i]+dist[i][d]` over all possible $i \in [0..V\text{-}1]$. 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 `AM` and then run the following four-liner code with three nested loops shown below. When it terminates, `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_warshall.cpp|java|py|ml`

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[23]. 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 `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?

---

[23]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).
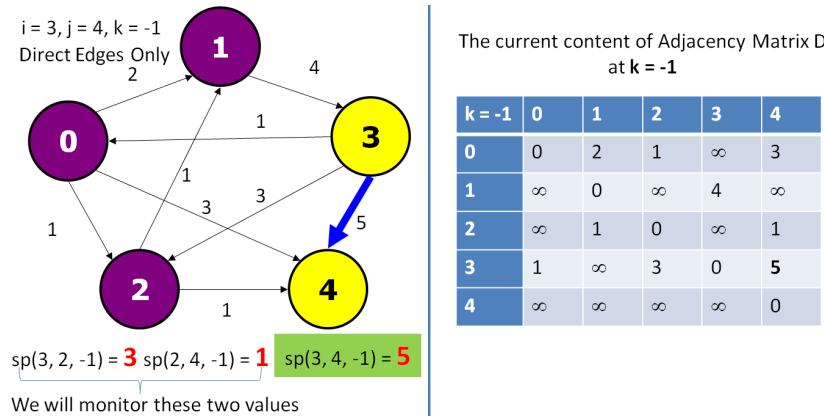
Figure 4.23: Floyd-Warshall Explanation 1

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) = 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.
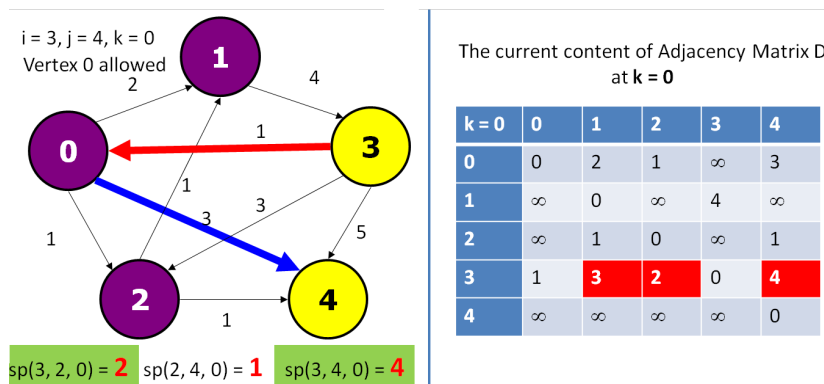
Figure 4.24: Floyd-Warshall Explanation 2

Floyd-Warshall then gradually allow $k = 0$, then $k = 1$, $k = 2$ ..., 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 $k = 0$, `sp(3,2,0)`—which we will need later—also drop from 3 to `sp(3,0,-1)` + `sp(0,2,-1)` = 1+1 = 2. Floyd-Warshall will process `sp(i,j,0)` for all other pairs considering only vertex 0 as the intermediate vertex but there is only one more change: `sp(3,1,0)` from ∞ down to 3.
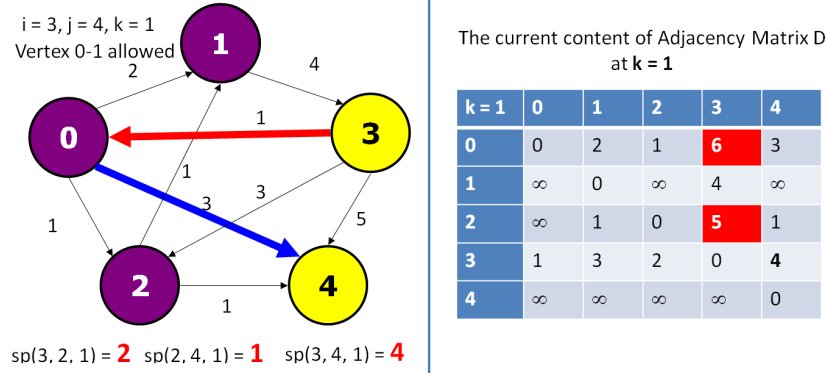
Figure 4.25: Floyd-Warshall Explanation 3

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 `sp(3,2,1)`, `sp(2,4,1)`, nor to `sp(3,4,1)`. However, two other values change: `sp(0,3,1)` and `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.
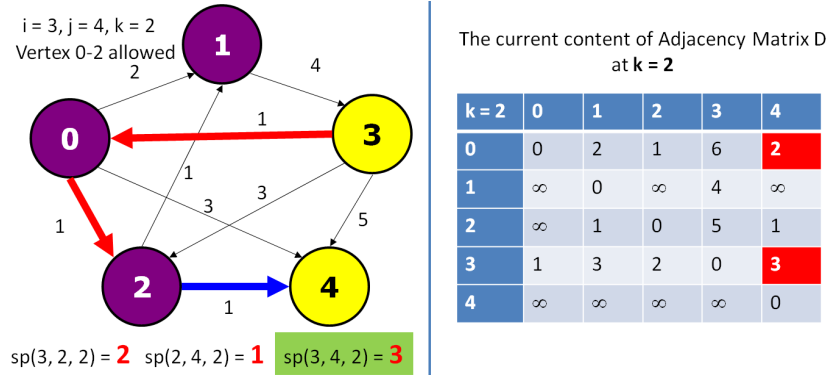
Figure 4.26: Floyd-Warshall Explanation 4

When we allow $k = 2$, i.e., vertex 0, 1, and 2 now can be used as the intermediate vertices, then `sp(3,4,2)` is reduced again as `sp(3,4,2)` = `sp(3,2,2)+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 `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:

$D_{i,j}^{-1} = weight(i,j)$. This is the base case when we do not use any intermediate vertices.
$D_{i,j}^k = min(D_{i,j}^{k-1}, D_{i,k}^{k-1} + D_{k,j}^{k-1}) = min(\text{not using vertex } k, \text{using vertex } k)$, 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 $O(V^3)$.

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 `vi 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 i = 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] = 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 relax 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] = 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 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` ∈ `[0..V-1]`. If `AM[i][j] && 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[24]), 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

---

[24]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 ≈ 1s time limit, 1 test case only.

Graph Criteria	BFS $O(V + E)$	Dijkstra's $O((V+E)\log V)$	Bellman-Ford $O(VE)$	Floyd-Warshall $O(V^3)$
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[25] 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].
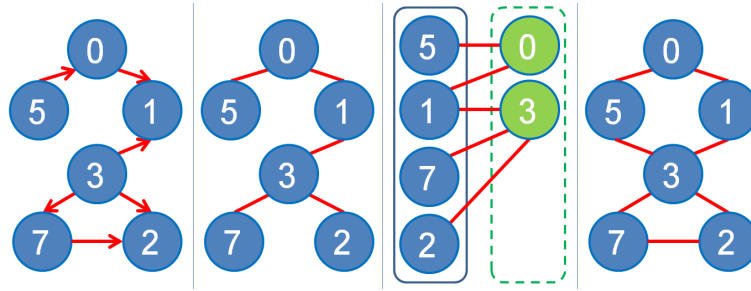
Figure 4.28: Special Graphs (L-to-R): DAG, Tree, Bipartite Graph, Eulerian Graph

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)[26] *Longest Paths* problem is a problem of finding the longest (simple[27]) paths from a starting vertex s to other vertices. The decision version of this problem

---

[25]There are a few other rare special graphs (see Section 4.6.5).

[26]Actually this can be multi-sources, as we can start from any vertex with 0 incoming degree.

[27]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[28]. 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[29] 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 $0 \to 1 \to 2 \to 4 \to 5$ 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.
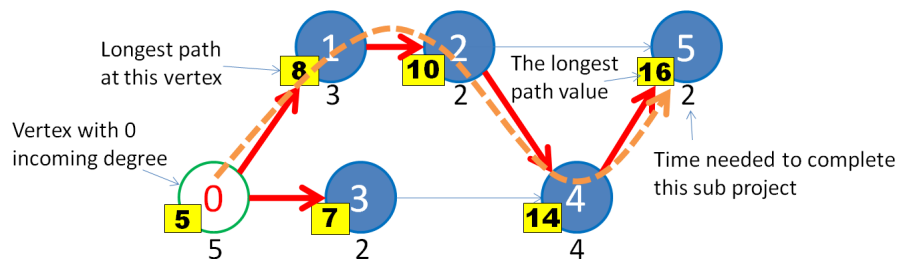
Figure 4.29: The Longest Path on this DAG

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).
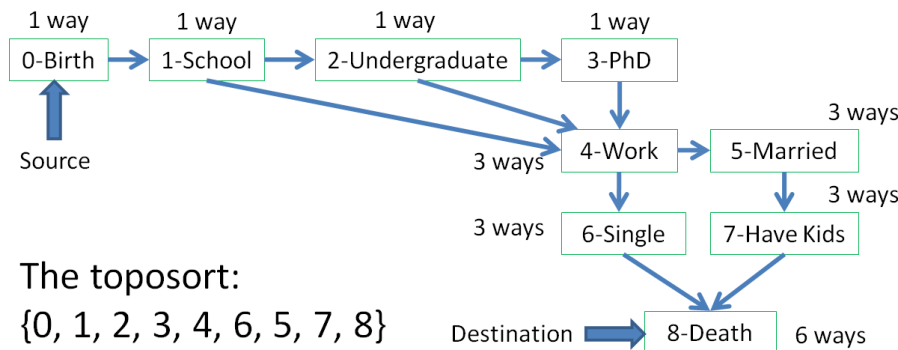
Figure 4.30: Example of Counting Paths in DAG - Bottom-Up

---

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'.

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

[29]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.
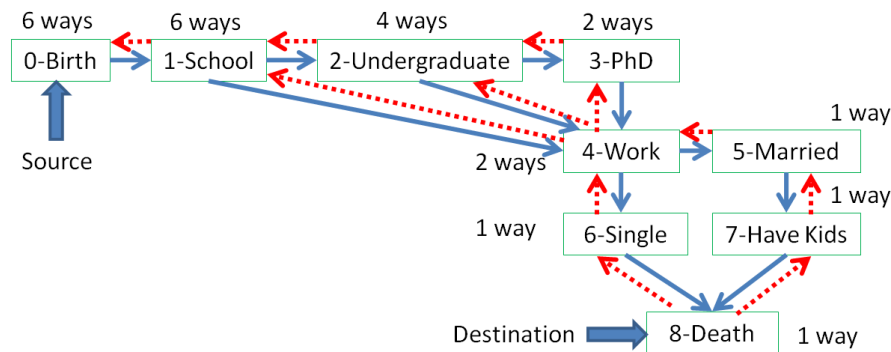
Figure 4.31: Example of Counting Paths in DAG - Top-Down

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.
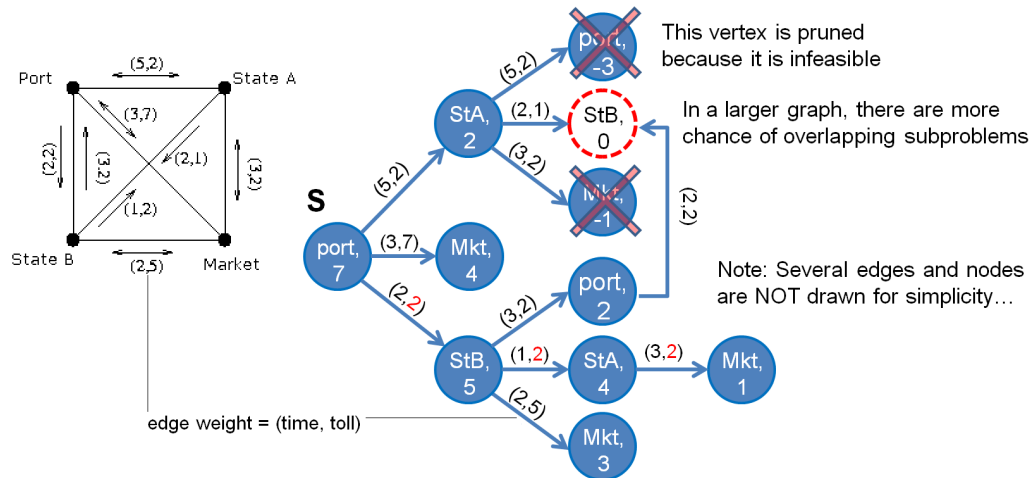
Figure 4.32: The Given General Graph (left) is Converted to DAG

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_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 (`port, t`) in the DAG. Every time the fishmonger moves from a current city `cur` to another city `X`, we move to a modified vertex (`X, t_left-travelTime[cur][X]`) in the DAG via edge with weight `toll[cur][X]`. As time is a diminishing resource, we will never encounter a cyclic situation. We can then use this (Top-Down) DP recurrence: `dp(cur, t_left)` to find the shortest path (in terms of total tolls paid) on this DAG. The answer can be found by calling `dp(0, t)`. The C++ code of `dp(cur, t_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 $O(n^2t)$—do-able.

```
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.
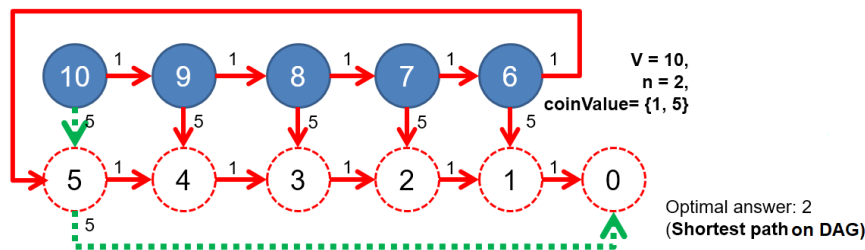
Figure 4.33: COIN-CHANGE as Shortest Paths on DAG

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., $\{10, 9, 8, \ldots, 1, 0\}$ 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: $10 \to 5 \to 0$ 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: $10 \to 5 \to 0$.
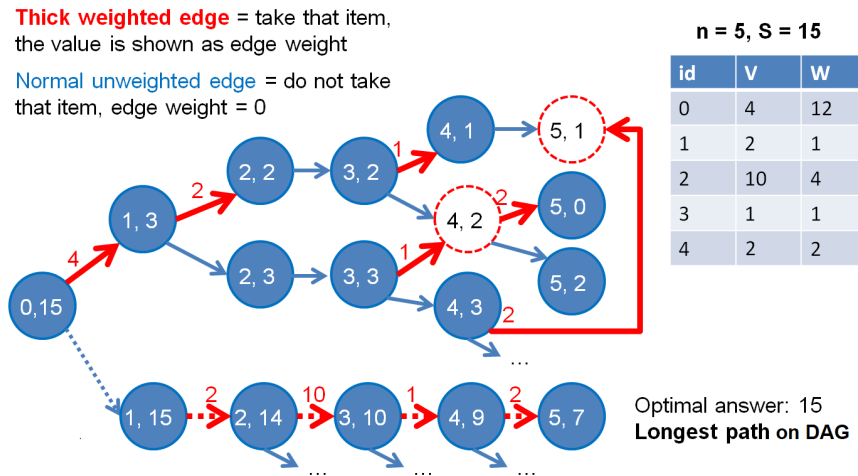
Figure 4.34: 0-1 KNAPSACK as Longest Paths on DAG

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`) $\to$ (`1, 15`) $\to$ (`2, 14`) $\to$ (`3, 10`) $\to$ (`4, 9`) $\to$ (`5, 7`) with value $0 + 2 + 10 + 1 + 2 = 15$.
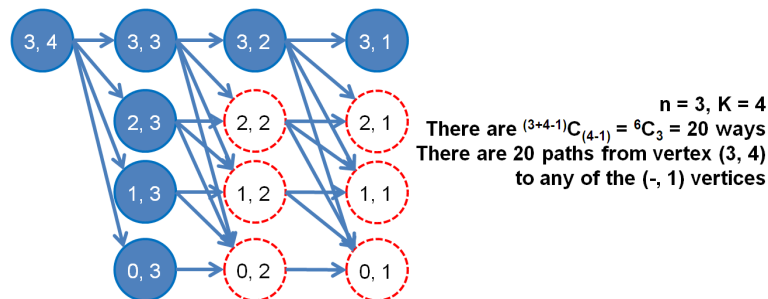
Figure 4.35: UVa 10943 as Counting Paths in DAG

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[30] 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) ≈ shortest paths on the implicit DAG, Longest Increasing Subsequence (LIS) ≈ 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)
```

---

[30]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).
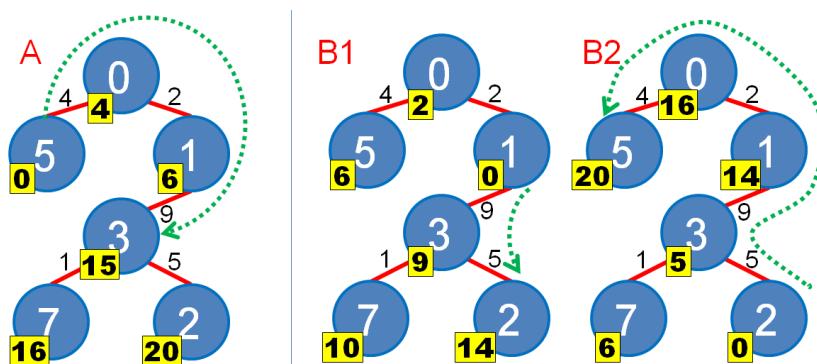
Figure 4.36: A: SSSP (Part of APSP); B1-B2: Diameter of Tree

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 $N[0]$), $_{48}C_2$ 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 → j` if `set1[i] + set2[j]` is prime, see Figure 4.37—left.
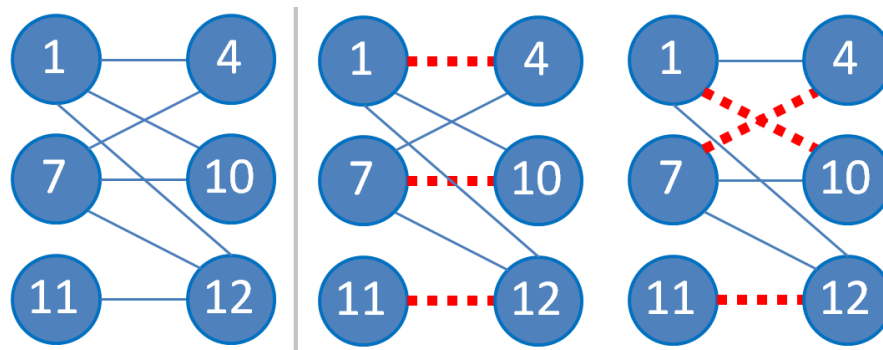
Figure 4.37: Bipartite Matching problem

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[31]' 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.

---

[31]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.
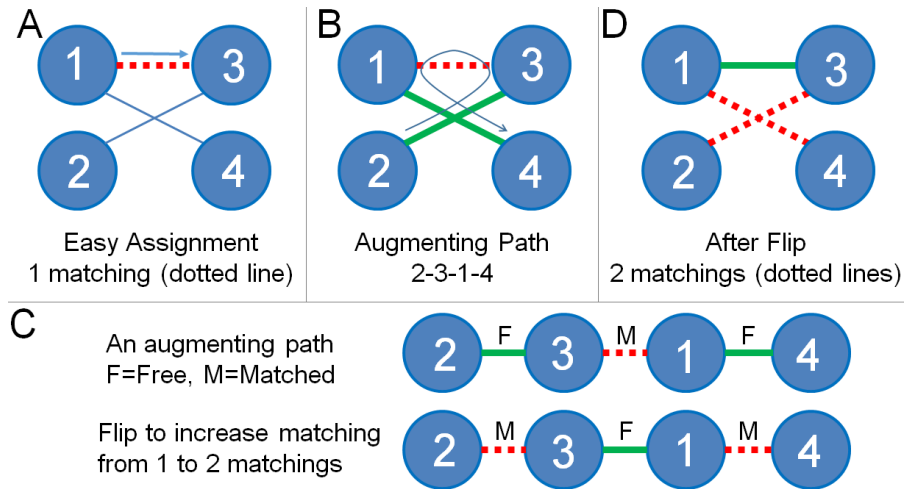
Figure 4.38: Augmenting Path Algorithm

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[32] code V times, it runs in $O(VE)$. The short implementation code is shown below.

```cpp
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
}
```

```cpp
// 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";
```

---

[32]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[33]. 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*[34] in a graph is a trail[35] 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.
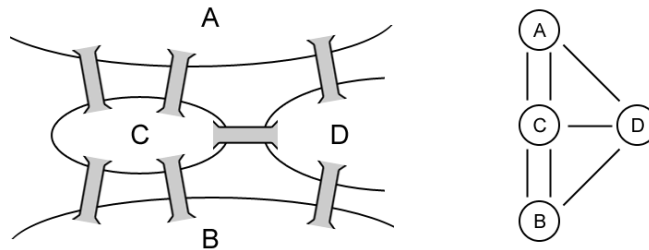
Figure 4.39: Königsberg bridges problem and its corresponding graph representation.

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.

---

[33]These better algorithms will be discussed in Book 2.

[34]Although the name is Eulerian **path**, it actually is a trail in graph theory.

[35]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 $u \to v$ **or** $v \to u$. 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.
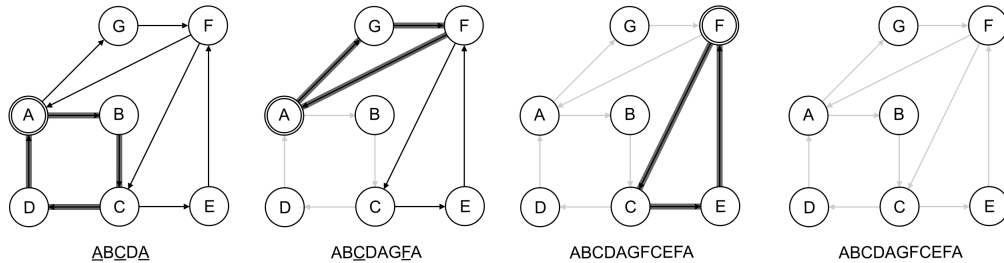
Figure 4.40: Example of Hierholzer's algorithm. The underlined characters represent vertices which still have incident edges which are not yet part of the trail.

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[36].

```
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`

---

[36]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.
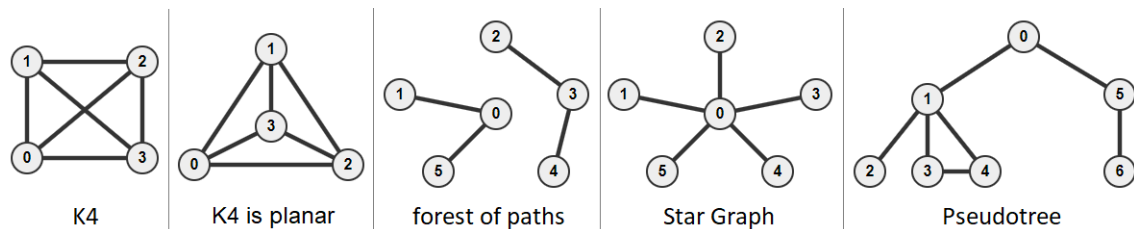
Figure 4.41: A Few Other Special Graphs

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[37]/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).

---

[37]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[38]

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 ≤ 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 traversal)
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*.

---

[38]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: $v_0 \to v_1 \to v_2 \to ... \to v_{2k-1} \to v_{2k} \to v_0$. 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 $\to$ 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 = $\{0 \to 1, 1 \to 2, 0 \to 2\}$. If we don't use the third DFS state EXPLORED, we will accidentally classify edge $0 \to 2$ 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 (`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 `AM[i][k]+AM[k][j]` which will *overflow* if both `AM[i][k]` and `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 $1\,061\,109\,567$ – 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:
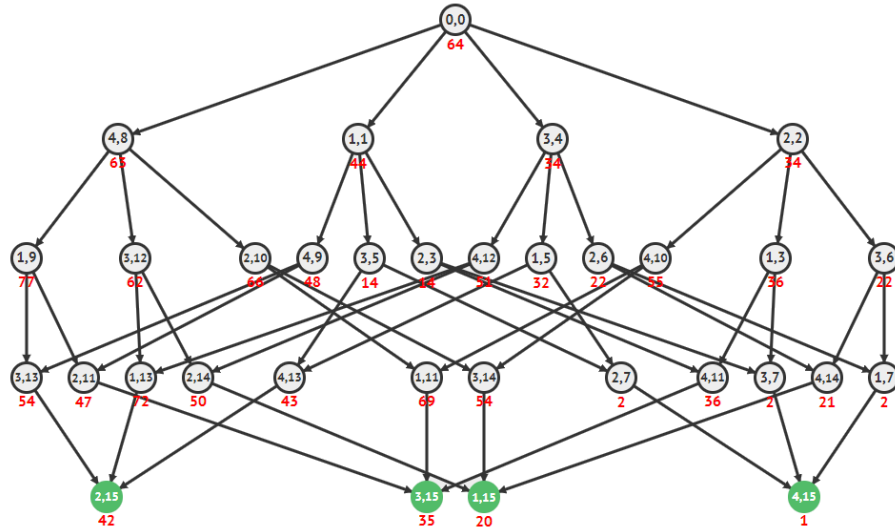
Figure 4.42: Recursion DAG of TSP with $n = 5$; Also See Figure 3.2
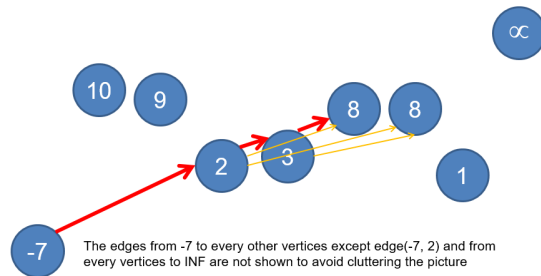
Figure 4.43: LIS as Longest Path of an Implicit DAG; Also See Figure 3.13

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[39], 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[40] 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	**Graph Traversal**	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	**Special Graphs**	113	≈ 26%	≈ 3.3%
	Total	431		≈ 12.5%

---

[39]In CP4, we move Network Flow section that was previously in this chapter (in CP3) to Book 2.

[40]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). "`RACECAR`".

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 $\pi \times \int_0^2 (e^{-x^2} + 2 \cdot \sqrt{x})^2$?

   > **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 15×15 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). $2\,279\,184$ 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/ioi-syllabus/ioi-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&pm=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] Rujia Liu. *Algorithm Contests for Beginners (In Chinese).* Tsinghua University Press, 2009.

[40] Rujia 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/.

[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

295