Copyrighted Material

ELEMENTS *of* PROGRAMMING INTERVIEWS

ADNAN **AZIZ** | AMIT **PRAKASH** | TSUNG-HSIEN **LEE**

Copyrighted Material

Table of Contents

I The Interview **5**

1 Getting Ready · 6

2 Strategies For A Great Interview · 11

3 Conducting An Interview · 18

4 Problem Solving Patterns · 22

II Problems **46**

5 Primitive Types · 47

6 Arrays and Strings · 52

7 Linked Lists · 62

8 Stacks and Queues · 67

9 Binary Trees · 73

10 Heaps · 80

11 Searching · 84

12 Hash Tables · 92

13 Sorting · 98

14 Binary Search Trees · 104

15 Meta-algorithms · 114

16 Algorithms on Graphs · 130

17 Intractability · 138

18 Parallel Computing · 144

19 Design Problems · 150

20 Probability · 155

21 Discrete Mathematics · 163

III Solutions **171**

IV Notation and Index **469**

Index of Terms · 472

Introduction

And it ought to be remembered that there is nothing more difficult to take in hand, more perilous to conduct, or more uncertain in its success, than to take the lead in the introduction of a new order of things.

— N. MACHIAVELLI, 1513

Elements of Programming Interviews (EPI) aims to help engineers interviewing for software development positions. The primary focus of EPI is data structures, algorithms, system design, and problem solving. The material is largely presented through questions.

An interview problem

Let's begin with Figure 1 below. It depicts movements in the share price of a company over 40 days. Specifically, for each day, the chart shows the daily high and low, and the price at the opening bell (denoted by the white square). Suppose you were asked in an interview to design an algorithm that determines the maximum profit that could have been made by buying and then selling a single share over a given day range, subject to the constraint that the buy and the sell have to take place at the start of the day. (This algorithm may be needed to backtest a trading strategy.)

You may want to stop reading now, and attempt this problem on your own.

First clarify the problem. For example, you should ask for the input format. Let's say the input consists of three arrays L, H, and S, of nonnegative floating point numbers, representing the low, high, and starting prices for each day. The constraint that the purchase and sale have to take place at the start of the day means that it

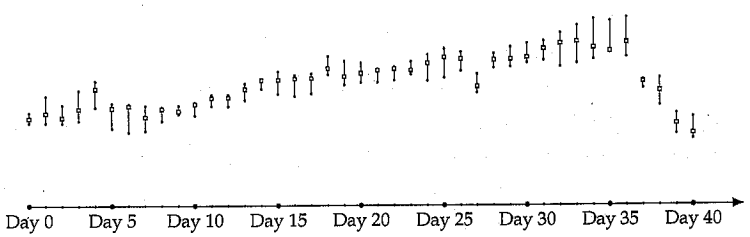


Figure 1: Share price as a function of time.

suffices to consider S. You may be tempted to simply return the difference of the minimum and maximum elements in S. If you try a few test cases, you will see that the minimum can occur after the maximum, which violates the requirement in the problem statement—you have to buy before you can sell.

At this point, a brute-force algorithm would be appropriate. For each pair of indices i and $j > i$ compute $p_{i,j} = S[j] - S[i]$ and compare this difference to the largest difference, d, seen so far. If $p_{i,j}$ is greater than d, set d to $p_{i,j}$. You should be able to code this algorithm using a pair of nested for-loops and test it in a matter of a few minutes. You should also derive its time complexity as a function of the length n of the input array. The inner loop is invoked $n - 1$ times, and the i-th iteration processes $n - 1 - i$ elements. Processing an element entails computing a difference, performing a compare, and possibly updating a variable, all of which take constant time. Hence the run time is proportional to $\sum_{k=0}^{n-2}(n-1-k) = \frac{(n-1)(n)}{2}$, i.e., the time complexity of the brute-force algorithm is $O(n^2)$. You should also consider the space complexity, i.e., how much memory your algorithm uses. The array itself takes memory proportional to n, and the additional memory used by the brute-force algorithm is a constant independent of n—a couple of iterators and one temporary floating point variable.

Once you have a working algorithm, try to improve upon it. Specifically, an $O(n^2)$ algorithm is usually not acceptable when faced with large arrays. You may have heard of an algorithm design pattern called divide and conquer. It yields the following algorithm for this problem. Split S into two subarrays, $S[0 : \lfloor \frac{n}{2} \rfloor]$ and $S[\lfloor \frac{n}{2} \rfloor + 1 : n - 1]$; compute the best result for the first and second subarrays; and combine these results. In the combine step we take the better of the results for the two subarrays. However, we also need to consider the case where the optimum buy and sell take place in separate subarrays. When this is the case, the buy must be in the first subarray, and the sell in the second subarray, since the buy must happen before the sell. If the optimum buy and sell are in different subarrays, the optimum buy price is the minimum price in the first subarray, and the optimum sell price is in the maximum price in the second subarray. We can compute these prices in $O(n)$ time with a single pass over each subarray. Therefore the time complexity $T(n)$ for the divide and conquer algorithm satisfies the recurrence relation $T(n) = 2T(\frac{n}{2}) + O(n)$, which solves to $O(n \log n)$.

The divide and conquer algorithm is elegant and fast. Its implementation entails some corner cases, e.g., an empty subarray, subarrays of length one, and an array in which the price decreases monotonically, but it can still be written and tested by a good developer in 20–30 minutes.

Looking carefully at the combine step of the divide and conquer algorithm, you may have a flash of insight. Specifically, you may notice that the maximum profit that can be made by selling on a specific day is determined by the minimum of the stock prices over the previous days. Since the maximum profit corresponds to selling on *some* day, the following algorithm correctly computes the maximum profit. Iterate through S, keeping track of the minimum element m seen thus far. If the difference of the current element and m is greater than the maximum profit recorded so far, update

the maximum profit. This algorithm performs a constant amount of work per array element, leading to an $O(n)$ time complexity. It uses two float-valued variables (the minimum element and the maximum profit recorded so far) and an iterator, i.e., $O(1)$ additional space. It is considerably simpler to implement than the divide and conquer algorithm—a few minutes should suffice to write and test it. Working code is presented in Solution 6.3 on Page 185.

If in a 45–60 minutes interview, you can develop the algorithm described above, implement and test it, and analyze its complexity, you would have had a very successful interview. In particular, you would have demonstrated to your interviewer that you possess several key skills:

- The ability to rigorously formulate real-world problems.
- The skills to solve problems and design algorithms.
- The tools to go from an algorithm to a tested program.
- The analytical techniques required to determine the computational complexity of your solution.

Book organization and study guide

Interviewing successfully is about more than being able to intelligently select data structures and design algorithms quickly. For example, you also need to know how to identify suitable companies, pitch yourself, ask for help when you are stuck on an interview problem, and convey your enthusiasm. These aspects of interviewing are the subject of Chapters 1–3, and are summarized in Table 1.1 on Page 7.

Chapter 1 is specifically concerned with preparation; Chapter 2 discusses how you should conduct yourself at the interview itself; and Chapter 3 describes interviewing from the interviewer's perspective. The latter is important for candidates too, because of the insights it offers into the decision making process. Chapter 4 reviews problem solving patterns.

Since not everyone will have the time to work through EPI in its entirety, we have prepared a study guide (Table 1.2 on Page 8) to problems you should solve, based on the amount of time you have available.

The problem chapters are organized as follows. Chapters 5–14 are concerned with basic data structures, such as arrays and binary search trees, and basic algorithms, such as binary search and quicksort. In our experience, this is the material that most interview questions are based on. Chapters 15–17 cover advanced algorithm design principles, such as dynamic programming and heuristics, as well as graphs. Chapters 18–19 focus on distributed and parallel programming, and design problems. Chapters 20–21 study probability and discrete mathematics; candidates for positions in finance companies should pay special attention to them.

The notation, specifically the symbols we use for describing algorithms, e.g., $|S|, A[i:j]$, is fairly standard. It is summarized starting on Page 470; you are strongly recommended to review it. Terms, e.g., BFS and dequeue, are indexed starting on Page 471.

Problems, solutions, variants, and ninjas

Most solutions in EPI are based on basic concepts, such as arrays, hash tables, and binary search, used in clever ways. A few solutions use relatively advanced machinery, e.g., Dijkstra's shortest path algorithm or random variables. You will encounter such problems in an interview only if you have a graduate degree or claim specialized knowledge, such as graph theory or randomized algorithms.

Most solutions include code snippets. These are primarily written in C++, and use C++11 features. Programs concerned with concurrency are in Java. C++11 features germane to EPI are reviewed on Page 172. A guide to reading C++ programs for Java developers is given on Page 172. Source code, which includes randomized and directed test cases, can be found at `ElementsOfProgrammInginterviews.com/code`. System design problems, and some problems related to probability and discrete mathematics, are conceptual and not meant to be coded.

At the end of many solutions we outline problems that are related to the original question. We classify such problems as variants and ϵ-variants. A variant is a problem whose formulation or solution is similar to the solved problem. An ϵ-variant is a problem whose solution differs slightly, if at all, from the given solution. Some ϵ-variants may be phrased quite differently from the original problem.

Approximately a quarter of the questions in EPI have a white ninja ( ) or black ninja ( ) designation. White ninja problems are more challenging, and are meant for applicants from whom the bar is higher, e.g., graduate students and tech leads. Black ninja problems are exceptionally difficult, and are suitable for testing a candidate's response to stress, as described on Page 16. Non-ninja questions should be solvable within an hour-long interview and, in some cases, take substantially less time.

Level and prerequisites

We expect readers to be familiar with data structures and algorithms taught at the undergraduate level. The chapters on concurrency and system design require knowledge of locks, distributed systems, operating systems (OS), and insight into commonly used applications. Much of the material in the chapters on meta-algorithms, graphs, intractability, probability, and discrete mathematics is more advanced and geared towards candidates with graduate degrees or specialized knowledge.

The review at the start of each chapter is not meant to be comprehensive and if you are not familiar with the material, you should first study it in an algorithms textbook. There are dozens of such texts and our preference is to master one or two good books rather than superficially sample many. We like *Algorithms* by Dasgupta, Papadimitriou, and Vazirani because it is succinct and beautifully written; *Introduction to Algorithms* by Cormen, Leiserson, Rivest, and Stein is more detailed and serves as a good reference.

Since our focus is on problems that can be solved in an interview, we do not include many elegant algorithm design problems. Similarly, we do not have any straightforward review problems; you may want to brush up on these using textbooks.

Part I

The Interview

CHAPTER

1 Getting Ready

Before everything else, getting ready is the secret of success.

— H. FORD

The most important part of interview preparation is knowing the material and practicing problem solving. However the nontechnical aspects of interviewing are also very important, and often overlooked. Chapters 1–3 are concerned with the nontechnical aspects of interviewing, ranging from résumé preparation to how hiring decisions are made. These aspects of interviewing are summarized in Table 1.1 on the facing page

Study guide

Ideally, you would prepare for an interview by solving all the problems in EPI. This is doable over 12 months if you solve a problem a day, where solving entails writing a program and getting it to work on some test cases.

Since different candidates have different time constraints, we have outlined several study scenarios, and recommended a subset of problems for each scenario. This information is summarized in Table 1.2 on Page 8. The preparation scenarios we consider are Hackathon (a weekend entirely devoted to preparation), finals cram (one week, 3–4 hours per day), term project (four weeks, 1.5–2.5 hours per day), and algorithms class (3–4 months, 1 hour per day).

At Google, Amazon, Microsoft, and similar companies, a large majority of the interview questions are drawn from the topics in Chapters 5–14. Exercise common sense when using Table 1.2, e.g., if you are interviewing for a position with a financial firm, you should pay more emphasis to Probability and Discrete Mathematics. If you have a graduate degree or are interviewing for a lead position, add some starred problems.

Although an interviewer may occasionally ask a question directly from EPI, you should not base your preparation on memorizing solutions. Rote learning will likely lead to your giving a perfect solution to the wrong problem.

Table 1.1: A summary of nontechnical aspects of interviewing

The Interview Lifecycle, on the current page
- Identify companies, contacts
- Résumé preparation
  - Basic principles
  - Website with links to projects
  - LinkedIn profile & recommendations
- Résumé submission
- Mock interview practice
- Phone/campus screening
- On-site interview
- Negotiating an offer

At the Interview, on Page 11
- Don't solve the wrong problem
- Get specs & requirements
- Construct sample input/output
- Work on small examples first
- Spell out the brute-force solution
- Think out loud
- Apply patterns
- Test for corner-cases
- Use proper syntax
- Manage the whiteboard
- Be aware of memory management
- Get function signatures right

General Advice, on Page 15
- Know the company & interviewers
- Communicate clearly
- Be passionate
- Be honest
- Stay positive
- Don't apologize
- Be well-groomed
- Mind your body language
- Leave perks and money out
- Be ready for a stress interview
- Learn from bad outcomes
- Negotiate the best offer

Conducting an Interview, on Page 18
- Don't be indecisive
- Create a brand ambassador
- Coordinate with other interviewers
  - know what to test on
  - look for patterns of mistakes
- Characteristics of a good problem:
  - no single point of failure
  - has multiple solutions
  - covers multiple areas
  - is calibrated on colleagues
  - does not require unnecessary domain knowledge
- Control the conversation
  - draw out quiet candidates
  - manage verbose/overconfident candidates
- Use a process for recording & scoring
- Determine what training is needed
- Apply the litmus test

The interview lifecycle

Generally speaking, interviewing takes place in the following steps:

1. Identify companies that you are interested in, and, ideally, find people you know at these companies.
2. Prepare your résumé using the guidelines on the following page, and submit it via a personal contact (preferred), or through an online submission process or a campus career fair.
3. Perform an initial phone screening, which often consists of a question-answer session over the phone or video chat with an engineer. You may be asked to submit code via a shared document or an online coding site such as ideone.com or collabedit.com. Don't take the screening casually—it can be extremely challenging.
4. Go for an on-site interview—this consists of a series of one-on-one interviews

Table 1.2: First read Chapter 4. For each chapter, first read its introductory text. Use textbooks for reference only. Unless a problem is italicized, it entails writing code. For Scenario i, write and test code for the problems in Columns 0 to $i-1$, and pseudo-code for the problems in Column i.

Scenario 1 Hackathon 3 days		Scenario 2 Finals cram 7 days	Scenario 3 Term project 1 month	Scenario 4 Algorithms class 4 months
C0	**C1**	**C2**	**C3**	**C4**
5.1	5.2, 5.5–5.6	5.7–5.8	5.3, 5.11–5.12	5.4, 5.13
6.1	6.14, 6.22	6.3, 6.9	6.15, 6.19, 6.21	6.17–6.18, 6.12, 6.23
7.1	7.2, 7.4	7.5–7.6 7.9	7.7–7.8	7.3, 7.10–7.11
8.1	8.3, 8.5, 8.9	8.10, 8.12	8.4, 8.6, 8.14	8.2, 8.8
9.5	9.7, 9.12–9.13	9.2, 9.8	9.6, 9.9	9.14
10.1	10.6, 10.8	10.2, 10.5	14.16–10.3	10.7, 10.9
11.2	11.3, 11.13	11.1, 11.9	11.5, 11.15	11.7, 11.12
12.9	12.2, 12.7	12.1, 12.8	*12.5, 12.13*	14.17, *12.16*
13.5	13.2, 13.12	*13.1*, 13.6, 13.10	*18.14*, 13.7	13.8, 13.14
14.1	14.5, 14.12	14.4, 14.7	14.11, 14.22	14.6, 14.13, 14.21
15.11	15.4, 15.22	15.2, 15.12, 15.26	15.10, 15.15	15.14, 15.25, *15.27*
16.1	16.7, *16.10*	16.3, 16.6	16.5, 16.9	16.11, *16.13*
17.2	17.8, 17.12	17.4, 17.10	17.1, 17.9	17.3, *17.11*
18.5	18.1, 18.8	18.3–18.4	18.2, 18.6	18.11, *18.15*
19.1	*19.4*, 19.7	*19.12, 19.15*	*19.2, 19.5, 19.8*	*19.6, 19.13*
20.6	20.3, 20.5	20.2, 20.7	20.1, 20.4	*20.12, 20.19*
21.1	*21.2, 21.4*	*21.5–21.6*	*21.9–21.10*	*21.7–21.8, 21.17*

with engineers and managers, and a conversation with your Human Resources (HR) contact.

5. Receive offers—these are usually a starting point for negotiations.

Note that there may be variations—e.g., a company may contact you, or you may submit via your college's career placement center. The screening may involve a homework assignment to be done before or after the conversation. The on-site interview may be conducted over a video chat session. Most on-sites are half a day, but others may last the entire day. For anything involving interaction over a network, be absolutely sure to work out logistics (a quiet place to talk with a landline rather than a mobile, familiarity with the coding website and chat software, etc.) well in advance.

We recommend that you interview at as many places as you can without it taking away from your job or classes. The experience will help you feel more comfortable with interviewing and you may discover you really like a company that you did not know much about.

The résumé

It always astonishes us to see candidates who've worked hard for at least four years

in school, and often many more in the workplace, spend 30 minutes jotting down random factoids about themselves and calling the result a résumé.

A résumé needs to address HR staff, the individuals interviewing you, and the hiring manager. The HR staff, who typically first review your résumé, look for keywords, so you need to be sure you have those covered. The people interviewing you and the hiring manager need to know what you've done that makes you special, so you need to differentiate yourself.

Here are some key points to keep in mind when writing a résumé:

1. Have a clear statement of your objective; in particular, make sure that you tailor your résumé for a given employer.
   - E.g., "My outstanding ability is developing solutions to computationally challenging problems; communicating them in written and oral form; and working with teams to implement them. I would like to apply these abilities at XYZ."
2. The most important points—the ones that differentiate you from everyone else—should come first. People reading your résumé proceed in sequential order, so you want to impress them with what makes you special early on. (Maintaining a logical flow, though desirable, is secondary compared to this principle.)
   - As a consequence, you should not list your programming languages, coursework, etc. early on, since these are likely common to everyone. You should list significant class projects (this also helps with keywords for HR.), as well as talks/papers you've presented, and even standardized test scores, if truly exceptional.
3. The résumé should be of a high-quality: no spelling mistakes; consistent spacings, capitalizations, numberings; and correct grammar and punctuation. Use few fonts. Portable Document Format (PDF is preferred, since it renders well across platforms.
4. Include contact information, a LinkedIn profile, and, ideally, a URL to a personal homepage with examples of your work. These samples may be class projects, a thesis, and links to companies and products you've worked on. Include design documents as well as a link to your version control repository.
5. If you can work at the company without requiring any special processing (e.g., if you have a Green Card, and are applying for a job in the US), make a note of that.
6. Have friends review your résumé; they are certain to find problems with it that you missed. It is better to get something written up quickly, and then refine it based on feedback.
7. A résumé does not have to be one page long—two pages are perfectly appropriate. (Over two pages is probably not a good idea.)
8. As a rule, we prefer not to see a list of hobbies/extracurricular activities (e.g., "reading books", "watching TV", "organizing tea party activities") unless they are really different (e.g., "Olympic rower") and not controversial.

Whenever possible, have a friend or professional acquaintance at the company route your résumé to the appropriate manager/HR contact—the odds of it reaching the right hands are much higher. At one company whose practices we are familiar with, a résumé submitted through a contact is 50 times more likely to result in a hire than one submitted online. Don't worry about wasting your contact's time—employees often receive a referral bonus, and being responsible for bringing in stars is also viewed positively.

Mock interviews

Mock interviews are a great way of preparing for an interview. Get a friend to ask you questions (from EPI or any other source) and solve them on a whiteboard, with pen and paper, or on a shared document. Have your friend take notes and give you feedback, both positive and negative. Make a video recording of the interview. You will cringe as you watch it, but it is better to learn of your mannerisms beforehand. Also ask your friend to give hints when you get stuck. In addition to sharpening your problem solving and presentation skills, the experience will help reduce anxiety at the actual interview setting.

CHAPTER

2

Strategies For A Great Interview

The essence of strategy is choosing what not to do.

— M. E. PORTER

A typical one hour interview with a single interviewer consists of five minutes of introductions and questions about the candidate's résumé. This is followed by five to fifteen minutes of questioning on basic programming concepts. The core of the interview is one or two detailed design questions where the candidate is expected to present a detailed solution on a whiteboard, paper, or IDE. Depending on the interviewer and the question, the solution may be required to include syntactically correct code and tests.

Approaching the problem

No matter how clever and well prepared you are, the solution to an interview problem may not occur to you immediately. Here are some things to keep in mind when this happens.

Clarify the question: This may seem obvious but it is amazing how many interviews go badly because the candidate spends most of his time trying to solve the wrong problem. If a question seems exceptionally hard, you may have misunderstood it.

A good way of clarifying the question is to state a concrete instance of the problem. For example, if the question is "find the first occurrence of a number greater than k in a sorted array", you could ask "if the input array is $\langle 2, 20, 30 \rangle$ and k is 3, then are you supposed to return 1, the index of 20?" These questions can be formalized as unit tests.

Work on small examples: Consider Problem 21.1 on Page 163, which entails determining which of the 500 doors are open. This problem may seem difficult at first. However, if you start working out which doors are going to be open up to the fifth door, you will see that only Door 1 and Door 4 are open. This may suggest to you that the door is open only if its index is a perfect square. Once you have this epiphany, the proof of its correctness is straightforward. (Keep in mind this approach will not work for all problems you encounter.)

Spell out the brute-force solution: Problems that are put to you in an interview tend to have an obvious brute-force solution that has a high time complexity compared to more sophisticated solutions. For example, instead of trying to work out

a DP solution for a problem (e.g., for Problem 15.12 on Page 121), try all the possible configurations. Advantages to this approach include: (1.) it helps you explore opportunities for optimization and hence reach a better solution, (2.) it gives you an opportunity to demonstrate some problem solving and coding skills, and (3.) it establishes that both you and the interviewer are thinking about the same problem. Be warned that this strategy can sometimes be detrimental if it takes a long time describe the brute-force approach.

Think out loud: One of the worst things you can do in an interview is to freeze up when solving the problem. It is always a good idea to think out loud. On the one hand, this increases your chances of finding the right solution because it forces you to put your thoughts in a coherent manner. On the other hand, this helps the interviewer guide your thought process in the right direction. Even if you are not able to reach the solution, the interviewer will form some impression of your intellectual ability.

Apply patterns: Patterns—general reusable solutions to commonly occurring problems—can be a good way to approach a baffling problem. Examples include finding a good data structure, seeing if your problem is a good fit for a general algorithmic technique, e.g., divide and conquer, recursion, or dynamic programming, and mapping the problem to a graph. Patterns are described in much more detail in Chapter 4.

Presenting the solution

Once you have an algorithm, it is important to present it in a clear manner. Your solution will be much simpler if you use Java or C++, and take advantage of libraries such as Collections or Boost. However, it is far more important that you use the language you are most comfortable with. Here are some things to keep in mind when presenting a solution.

Libraries: Master the libraries, especially the data structures. Do not waste time and lose credibility trying to remember how to pass an explicit comparator to a BST constructor. Remember that a hash function should use exactly those fields which are used in the equality check. A comparison function should be transitive.

Focus on the top-level algorithm: It's OK to use functions that you will implement later. This will let you focus on the main part of the algorithm, will penalize you less if you don't complete the algorithm. (Hash, equals, and compare functions are good candidates for deferred implementation.) Specify that you will handle main algorithm first, then corner cases. Add TODO comments for portions that you want to come back to.

Manage the whiteboard: You will likely use more of the board than you expect, so start at the top-left corner. Have a system for abbreviating variables, e.g., declare `stackMax` and then use `sm` for short. Make use of functions—skip implementing anything that's trivial (e.g., finding the maximum of an array) or standard (e.g., a thread pool).

Test for corner cases: For many problems, your general idea may work for

most inputs but there may be pathological instances where your algorithm (or your implementation of it) fails. For example, your binary search code may crash if the input is an empty array; or you may do arithmetic without considering the possibility of overflow. It is important to systematically consider these possibilities. If there is time, write unit tests. Small, extreme, or random inputs make for good stimuli. Don't forget to add code for checking the result. Often the code to handle obscure corner cases may be too complicated to implement in an interview setting. If so, you should mention to the interviewer that you are aware of these problems, and could address them if required.

Syntax: Interviewers rarely penalize you for small syntax errors since modern integrated development environments (IDEs) excel at handling these details. However lots of bad syntax may result in the impression that you have limited coding experience. Once you are done writing your program, make a pass through it to fix any obvious syntax errors before claiming you are done.

Have a convention for identifiers, e.g., `i,j,k` for array indices, `A,B,C` for arrays, `hm` for `HashMap`, `s` for a `String`, `sb` for a `StringBuilder`, etc.

Candidates often tend to get function signatures wrong and it reflects poorly on them. For example, it would be an error to write a function in C that returns an array but not its size. In C++ it is important to know whether to pass parameters by value or by reference. Use `const` as appropriate.

Memory management: Generally speaking, it is best to avoid memory management operations all together. In C++, if you are using dynamic allocation consider using scoped pointers. The run time environment will automatically deallocate the object a scoped pointer points to when it goes out of scope. If you explicitly allocate memory, ensure that in every execution path, this memory is de-allocated. See if you can reuse space. For example, some linked list problems can be solved with $O(1)$ additional space by reusing existing nodes.

Know your interviewers & the company

It can help you a great deal if the company can share with you the background of your interviewers in advance. You should use search and social networks to learn more about the people interviewing you. Letting your interviewers know that you have researched them helps break the ice and forms the impression that you are enthusiastic and will go the extra mile. For fresh graduates, it is also important to think from the perspective of the interviewers as described in Chapter 3.

Once you ace your interviews and have an offer, you have an important decision to make—is this the organization where you want to work? Interviews are a great time to collect this information. Interviews usually end with the interviewers letting the candidates ask questions. You should make the best use of this time by getting the information you would need and communicating to the interviewer that you are genuinely interested in the job. Based on your interaction with the interviewers, you may get a good idea of their intellect, passion, and fairness. This extends to the team and company.

In addition to knowing your interviewers, you should know about the company vision, history, organization, products, and technology. You should be ready to talk about what specifically appeals to you, and to ask intelligent questions about the company and the job. Prepare a list of questions in advance; it gets you helpful information as well as shows your knowledge and enthusiasm for the organization. You may also want to think of some concrete ideas around things you could do for the company; be careful not to come across as a pushy know-it-all.

All companies want bright and motivated engineers. However, companies differ greatly in their culture and organization. Here is a brief classification.

Startup, e.g., Quora: values engineers who take initiative and develop products on their own. Such companies do not have time to train new hires, and tend to hire candidates who are very fast learners or are already familiar with their technology stack, e.g., their web application framework, machine learning system, etc.

Mature consumer-facing company, e.g., Google: wants candidates who understand emerging technologies from the user's perspective. Such companies have a deeper technology stack, much of which is developed in-house. They have the resources and the time to train a new hire.

Enterprise-oriented company, e.g., Oracle: looks for developers familiar with how large projects are organized, e.g., engineers who are familiar with reviews, documentation, and rigorous testing.

Government contractor, e.g., Lockheed-Martin: values knowledge of specifications and testing, and looks for engineers who are familiar with government-mandated processes.

Embedded systems/chip design company, e.g., National Instruments: wants software engineers who know enough about hardware to interface with the hardware engineers. The tool chain and development practices at such companies tend to be very mature.

General conversation

Often interviewers will ask you questions about your past projects, such as a senior design project or an internship. The point of this conversation is to answer the following questions:

Can the candidate clearly communicate a complex idea? This is one of the most important skills for working in an engineering team. If you have a grand idea to redesign a big system, can you communicate it to your colleagues and bring them on board? It is crucial to practice how you will present your best work. Being precise, clear, and having concrete examples can go a long way here. Candidates communicating in a language that is not their first language, should take extra care to speak slowly and make more use of the whiteboard to augment their words.

Is the candidate passionate about his work? We always want our colleagues to be excited, energetic, and inspiring to work with. If you feel passionately about your work, and your eyes light up when describing what you've done, it goes a long way in establishing you as a great colleague. Hence when you are asked to describe a

project from the past, it is best to pick something that you are passionate about rather than a project that was complex but did not interest you.

Is there a potential interest match with some project? The interviewer may gauge areas of strengths for a potential project match. If you know the requirements of the job, you may want to steer the conversation in that direction. Keep in mind that because technology changes so fast many teams prefer a strong generalist, so don't pigeonhole yourself.

Other advice

Be honest: Nobody wants a colleague who falsely claims to have tested code or done a code review. Dishonesty in an interview is a fast pass to an early exit.

Remember, nothing breaks the truth more than stretching it—you should be ready to defend anything you claim on your résumé. If your knowledge of Python extends only as far as having cut-and-paste sample code, do not add Python to your résumé.

Similarly, if you have seen a problem before, you should say so. (Be sure that it really is the same problem, and bear in mind you should describe a correct solution quickly if you claim to have solved it before.) Interviewers have been known to collude to ask the same question of a candidate to see if he tells the second interviewer about the first instance. An interviewer may feign ignorance on a topic he knows in depth to see if a candidate pretends to know it.

Keep a positive spirit: A cheerful and optimistic attitude can go a long way. Absolutely nothing is to be gained, and much can be lost, by complaining how difficult your journey was, how you are not a morning person, how inconsiderate the airline/hotel/HR staff were, etc.

Don't apologize: Candidates sometimes apologize in advance for a weak GPA, rusty coding skills, or not knowing the technology stack. Their logic is that by being proactive they will somehow benefit from lowered expectations. Nothing can be further from the truth. It focuses attention on shortcomings. More generally, if you do not believe in yourself, you cannot expect others to believe in you.

Appearance: Most software companies have a relaxed dress-code, and new graduates may wonder if they will look foolish by overdressing. The damage done when you are too casual is greater than the minor embarrassment you may feel at being overdressed. It is always a good idea to err on the side of caution and dress formally for your interviews. At the minimum, be clean and well-groomed.

Be aware of your body language: Think of a friend or coworker slouched all the time or absentmindedly doing things that may offend others. Work on your posture, eye contact and handshake, and remember to smile.

Keep money and perks out of the interview: Money is a big element in any job but it is best left discussed with the HR division after an offer is made. The same is true for vacation time, day care support, and funding for conference travel.

Stress interviews

Some companies, primarily in the finance industry, make a practice of having one of the interviewers create a stressful situation for the candidate. The stress may be injected technically, e.g., via a ninja problem, or through behavioral means, e.g., the interviewer rejecting a correct answer or ridiculing the candidate. The goal is to see how a candidate reacts to such situations—does he fall apart, become belligerent, or get swayed easily. The guidelines in the previous section should help you through a stress interview. (Bear in mind you will not know *a priori* if a particular interviewer will be conducting a stress interview.)

Learning from bad outcomes

The reality is that not every interview results in a job offer. There are many reasons for not getting a particular job. Some are technical: you may have missed that key flash of insight, e.g., the key to solving the maximum-profit on Page 1 in linear time. If this is the case, go back and solve that problem, as well as related problems.

Often, your interviewer may have spent a few minutes looking at your résumé—this is a depressingly common practice. This can lead to your being asked questions on topics outside of the area of expertise you claimed on your résumé, e.g., routing protocols or Structured Query Language (SQL). If so, make sure your résumé is accurate, and brush up on that topic for the future.

You can fail an interview for nontechnical reasons, e.g., you came across as uninterested, or you did not communicate clearly. The company may have decided not to hire in your area, or another candidate with similar ability but more relevant experience was hired.

You will not get any feedback from a bad outcome, so it is your responsibility to try and piece together the causes. Remember the only mistakes are the ones you don't learn from.

Negotiating an offer

An offer is not an offer till it is on paper, with all the details filled in. All offers are negotiable. We have seen compensation packages bargained up to twice the initial offer, but 10–20% is more typical. When negotiating, remember there is nothing to be gained, and much to lose, by being rude. (Being firm is not the same as being rude.)

To get the best possible offer, get multiple offers, and be flexible about the form of your compensation. For example, base salary is less flexible than stock options, sign-on bonus, relocation expenses, and Immigration and Naturalization Service (INS) filing costs. Be concrete—instead of just asking for more money, ask for a P% higher salary. Otherwise the recruiter will simply come back with a small increase in the sign-on bonus and claim to have met your request.

Your HR contact is a professional negotiator, whose fiduciary duty is to the company. He will know and use negotiating techniques such as reciprocity, getting consensus, putting words in your mouth ("don't you think that's reasonable?"), as

well as threats, to get the best possible deal for the company. (This is what recruiters themselves are evaluated on internally.) The Wikipedia article on negotiation lays bare many tricks we have seen recruiters employ.

One suggestion: stick to email, where it is harder for someone to paint you into a corner. If you are asked for something (such as a copy of a competing offer), get something in return. Often it is better to bypass the HR contact and speak directly with the hiring manager.

At the end of the day, remember your long term career is what counts, and joining a company that has a brighter future (social-mobile vs. legacy enterprise), or offers a position that has more opportunities to rise (developer vs. tester) is much more important than a 10–20% difference in compensation.

CHAPTER

3

Conducting An Interview

知己知彼，百戰不殆。
Translated—"If you know both yourself and your enemy, you can win numerous battles without jeopardy."

— *"The Art of War,"*
SUN TZU, 515 B.C.

In this chapter we review practices that help interviewers identify a top hire. We strongly recommend interviewees read it—knowing what an interviewer is looking for will help you present yourself better and increase the likelihood of a successful outcome.

For someone at the beginning of their career, interviewing may feel like a huge responsibility. Hiring a bad candidate is expensive for the organization, not just because the hire is unproductive, but also because he is a drain on the productivity of his mentors and managers, and sets a bad example. Firing someone is extremely painful as well as bad for to the morale of the team. On the other hand, discarding good candidates is problematic for a rapidly growing organization. Interviewers also have a moral responsibility not to unfairly crush the interviewee's dreams and aspirations.

Objective

The ultimate goal of any interview is to determine the odds that a candidate will be a successful employee of the company. The ideal candidate is smart, dedicated, articulate, collegial, and gets things done quickly, both as an individual and in a team. Ideally, your interviews should be designed such that a good candidate scores 1.0 and a bad candidate scores 0.0.

One mistake, frequently made by novice interviewers, is to be indecisive. Unless the candidate walks on water or completely disappoints, the interviewer tries not to make a decision and scores the candidate somewhere in the middle. This means that the interview was a wasted effort.

A secondary objective of the interview process is to turn the candidate into a brand ambassador for the recruiting organization. Even if a candidate is not a good fit for the organization, he may know others who would be. It is important for the candidate to have an overall positive experience during the process. It seems obvious

that it is a bad idea for an interviewer to check email while the candidate is talking or insult the candidate over a mistake he made, but such behavior is depressingly common. Outside of a stress interview, the interviewer should work on making the candidate feel positively about the experience, and, by extension, the position and the company.

What to ask

One important question you should ask yourself as an interviewer is how much training time your work environment allows. For a startup it is important that a new hire is productive from the first week, whereas a larger organization can budget for several months of training. Consequently, in a startup it is important to test the candidate on the specific technologies that he will use, in addition to his general abilities.

For a larger organization, it is reasonable not to emphasize domain knowledge and instead test candidates on data structures, algorithms, system design skills, and problem solving techniques. The justification for this is as follows. Algorithms, data structures, and system design underlie all software. Algorithms and data structure code is usually a small component of a system dominated by the user interface (UI), I/O, and format conversion. It is often hidden in library calls. However, such code is usually the crucial component in terms of performance and correctness, and often serves to differentiate products. Furthermore, platforms and programming languages change quickly but a firm grasp of data structures, algorithms, and system design principles, will always be a foundational part of any successful software endeavor. Finally, many of the most successful software companies have hired based on ability and potential rather than experience or knowledge of specifics, underlying the effectiveness of this approach to selecting candidates.

Most big organizations have a structured interview process where designated interviewers are responsible for probing specific areas. For example, you may be asked to evaluate the candidate on their coding skills, algorithm knowledge, critical thinking, or the ability to design complex systems. This book gives interviewers access to a fairly large collection of problems to choose from. When selecting a problem keep the following in mind:

No single point of failure—if you are going to ask just one question, you should not pick a problem where the candidate passes the interview if and only if he gets one particular insight. The best candidate may miss a simple insight, and a mediocre candidate may stumble across the right idea. There should be at least two or three opportunities for the candidates to redeem themselves. For example, problems that can be solved by dynamic programming can almost always be solved through a greedy algorithm that is fast but suboptimum or a brute-force algorithm that is slow but optimum. In such cases, even if the candidate cannot get the key insight, he can still demonstrate some problem solving abilities. Problem 6.3 on Page 53 exemplifies this type of question.

Multiple possible solutions—if a given problem has multiple solutions, the

chances of a good candidate coming up with a solution increases. It also gives the interviewer more freedom to steer the candidate. A great candidate may finish with one solution quickly enough to discuss other approaches and the trade-offs between them. For example, Problem 11.15 on Page 90 can be solved using a hash table or a bit array; the best solution makes use of binary search.

Cover multiple areas—even if you are responsible for testing the candidate on algorithms, you could easily pick a problem that also exposes some aspects of design and software development. For example, Problem 18.4 on Page 146 tests candidates on concurrency as well as data structures. Problem 17.1 on Page 139 requires knowledge of both dynamic programming and probability.

Calibrate on colleagues—interviewers often have an incorrect notion of how difficult a problem is for a thirty minute or one hour interview. It is a good idea to check the appropriateness of a problem by asking one of your colleagues to solve it and seeing how much difficulty they have with it.

No unnecessary domain knowledge—it is not a good idea to quiz a candidate on advanced graph algorithms if the job does not require it and the candidate does not claim any special knowledge of the field. (The exception to this rule is if you want to test the candidate's response to stress.)

Conducting the interview

Conducting a good interview is akin to juggling. At a high level, you want to ask your questions and evaluate the candidate's responses. Many things can happen in an interview that could help you reach a decision, so it is important to take notes. At the same time, it is important to keep a conversation going with the candidate and help him out if he gets stuck. Ideally, have a series of hints worked out beforehand, which can then be provided progressively as needed. Coming up with the right set of hints may require some thinking. You do not want to give away the problem, yet find a way for the candidate to make progress. Here are situations that may throw you off:

A candidate that gets stuck and shuts up: Some candidates get intimidated by the problem, the process, or the interviewer, and just shut up. In such situations, a candidate's performance does not reflect his true caliber. It is important to put the candidate at ease, e.g., by beginning with a straightforward question, mentioning that a problem is tough, or asking them to think out loud.

A verbose candidate: Candidates who go off on tangents and keep on talking without making progress render an interview ineffective. Again, it is important to take control of the conversation. For example you could assert that a particular path will not make progress.

An overconfident candidate: It is common to meet candidates who weaken their case by defending an incorrect answer. To give the candidate a fair chance, it is important to demonstrate to him that he is making a mistake, and allow him to correct it. Often the best way of doing this is to construct a test case where the candidate's solution breaks down.

Scoring and reporting

At the end of an interview, the interviewers usually have a good idea of how the candidate scored. However, is important to keep notes and revisit them before making a final decision. Whiteboard snapshots and samples of any code that the candidate wrote should also be recorded. You should standardize scoring based on which hints were given, how many questions the candidate was able to get to, etc. Although isolated minor mistakes can be ignored, sometimes when you look at all the mistakes together, clear signs of weakness in certain areas may emerge, such as a lack of attention to detail and unfamiliarity with a language.

When the right choice is not clear, wait for the next candidate instead of possibly making a bad hiring decision. The litmus test is to see if you would react positively to the candidate replacing a valuable member of your team.

CHAPTER

4 Problem Solving Patterns

It's not that I'm so smart, it's just that I stay with problems longer.

— A. EINSTEIN

Developing problem solving skills is like learning to play a musical instrument—books and teachers can point you in the right direction, but only your hard work will take you there. Just as a musician, you need to know underlying concepts, but theory is no substitute for practice.

Great problem solvers have skills that cannot be rigorously formalized. Still, when faced with a challenging programming problem, it is helpful to have a small set of "patterns"—general reusable solutions to commonly occurring problems—that may be applicable.

We now introduce several patterns and illustrate them with examples. We have classified these patterns into three categories:

- data structure patterns,
- algorithm design patterns, and
- abstract analysis patterns.

These patterns are summarized in Table 4.1 on the facing page, Table 4.2 on Page 28, and Table 4.3 on Page 38, respectively.

At a meta-level, concrete inputs are the best starting point for many problems. Small instances, such as an array or a BST containing 5–7 elements, specialized inputs, e.g., binary values, nonoverlapping intervals, connected graphs, etc., and extreme cases, for instance input that is sorted or contains duplicates, can offer tremendous insight.

The notion of patterns is very general; in particular, many patterns arise in the context of software design—the builder pattern, composition, publish-subscribe, etc. These are more suitable to large-scale systems, and as such are outside the scope of EPI, which is focused on smaller programs that can be solved in an interview.

Data structure patterns

A data structure is a particular way of storing and organizing related data items so that they can be manipulated efficiently. Usually the correct selection of data structures is key to designing a good algorithm. Different data structures are suited to different applications; some are highly specialized. For example, heaps are par-

ticularly well-suited for algorithms that merge sorted data streams, while compiler implementations usually use hash tables to look up identifiers.

Solutions often require a combination of data structures. Our solution to the problem of tracking the most visited pages on a website (Solution 14.18 on Page 110) involves a combination of a heap, a queue, a binary search tree, and a hash table.

Table 4.1: Data structure patterns.

Data structure	Key points
Primitive types	Know how `int`, `char`, `double`, etc. are represented in memory and the primitive operations on them.
Arrays & strings	Fast access for element at an index, slow lookups (unless sorted) and insertions. Be comfortable with notions of iteration, resizing, partitioning, merging, etc. Know how strings are represented in memory. Understand basic operators such as comparison, copying, matching, joining, splitting, etc.
Lists	Understand trade-offs with respect to arrays. Be comfortable with iteration, insertion, and deletion within singly and doubly linked lists. Know how to implement a list with dynamic allocation, and with arrays.
Stacks and queues	Understand insertion and deletion. Know array and linked list implementations.
Binary trees	Use for representing hierarchical data. Know about depth, height, leaves, search path, traversal sequences, successor/predecessor operations.
Heaps	Key benefit: $O(1)$ lookup find-max, $O(\log n)$ insertion, and $O(\log n)$ deletion of max. Node and array representations. Min-heap variant.
Hash tables	Key benefit: $O(1)$ insertions, deletions and lookups. Key disadvantages: not suitable for order-related queries; need for resizing; poor worst-case performance. Understand implementation using array of buckets and collision chains. Know hash functions for integers, strings, objects. Understand importance of equals function. Variants such as Bloom filters.
Binary search trees	Key benefit: $O(\log n)$ insertions, deletions, lookups, find-min, find-max, successor, predecessor when tree is balanced. Understand implementation using nodes and pointers. Be familiar with notion of balance, and operations maintaining balance. Know how to augment a binary search tree, e.g., interval trees and dynamic order statistics.

Primitive types

You should be comfortable with the basic types (chars, integers, doubles, etc.), their variants (unsigned, long, etc.), and operations on them (bitwise operators, comparison, etc.). Don't forget that the basic types differ among programming languages.

For example, Java has no unsigned integers, and the number of bits in an integer is compiler- and machine-dependent in C.

A common problem related to basic types is computing the number of bits set to 1 in an integer-valued variable x. To solve this problem you need to know how to manipulate individual bits in an integer. One straightforward approach is to iteratively test individual bits using an unsigned integer variable m initialized to 1. Iteratively identify bits of x that are set to 1 by examining the bitwise AND of m with x, shifting m left one bit at a time. The overall complexity is $O(n)$ where n is the length of the integer.

Another approach, which may run faster on some inputs, is based on computing $y = x \,\&\, !(x-1)$, where & is the bitwise AND operator. This is 1 at exactly the rightmost bit of x. Consequently, this bit may be removed from x by computing $x \oplus y$. The time complexity is $O(s)$, where s is the number of bits set to 1 in x.

In practice if the computation is done repeatedly, the most efficient approach would be to create a lookup table. In this case, we could use a 256 entry integer-valued array P such that $P[i]$ is the number of bits set to 1 in i. If x is 32 bits, the result can be computed by decomposing x into 4 disjoint bytes, $b3, b2, b1$, and $b0$. The bytes are computed using bitmasks and shifting, e.g., $b1$ is $(x \,\&\, 0xff00) \gg 8$. The final result is $P[b3] + P[b2] + P[b1] + P[b0]$. Computing the parity of an integer is closely related to counting the number of bits set to 1, and we present a detailed analysis of the parity problem in Solution 5.1 on Page 173.

Arrays and Strings

Conceptually, an array maps integers in the range $[0, n-1]$ to objects of a given type, where n is the number of objects in this array. Array lookup and insertion are fast, making arrays suitable for a variety of applications. Reading past the last element of an array is a common error, invariably with catastrophic consequences.

The following problem arises when optimizing quicksort: given an array A whose elements are comparable, and an index i, reorder the elements of A so that the initial elements are all less than $A[i]$, and are followed by elements equal to $A[i]$, which in turn are followed by elements greater than $A[i]$, using $O(1)$ space.

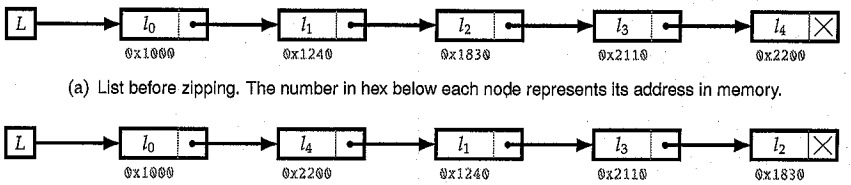
The key to the solution is to maintain two regions on opposite sides of the array that meet the requirements, and expand these regions one element at a time. Details are given in Solution 6.1 on Page 183.

Lists

An abstract data type (ADT) is a mathematical model for a class of data structures that have similar functionality. Strictly speaking, a list is an ADT, and not a data structure. It implements an ordered collection of values, which may be repeated. In the context of this book we view a list as a sequence of nodes where each node has a link to the next node in the sequence. In a doubly linked list each node additionally has a link to the prior node.

A list is similar to an array in that it contains objects in a linear order. The key differences are that inserting and deleting elements in a list has time complexity $O(1)$. On the other hand, obtaining the k-th element in a list is expensive, having $O(n)$ time complexity. Lists are usually building blocks of more complex data structures. However, they can be the subject of tricky problems in their own right, as illustrated by the following:

Given a singly linked list $\langle l_0, l_1, l_2, \ldots, l_{n-1} \rangle$, define the "zip" of the list to be $\langle l_0, l_{n-1}, l_1, l_{n-2}, \ldots \rangle$. Suppose you were asked to write a function that computes the zip of a list, with the constraint that it uses $O(1)$ space. The operation of this function is illustrated in Figure 4.1.



(a) List before zipping. The number in hex below each node represents its address in memory.

(b) List after zipping. Note that nodes are reused—no memory has been allocated.

Figure 4.1: Zipping a list.

The solution is based on an appropriate iteration combined with "pointer swapping", i.e., updating next field for each node. Refer to Solution 7.11 on Page 216 for details.

Stacks and queues

Stacks support last-in, first-out semantics for inserts and deletes, whereas queues are first-in, first-out. Both are ADTs, and are commonly implemented using linked lists or arrays. Similar to lists, stacks and queues are usually building blocks in a solution to a complex problem, but can make for interesting problems in their own right.

As an example consider the problem of evaluating Reverse Polish notation expressions, i.e., expressions of the form "$3, 4, \times, 1, 2, +, +$", "$1, 1, +, -2, \times$", or "$4, 6, /, 2, /$". A stack is ideal for this purpose—operands are pushed on the stack, and popped as operators are processed, with intermediate results being pushed back onto the stack. Details are given in Solution 8.2 on Page 221.

Binary trees

A binary tree is a data structure that is used to represent hierarchical relationships. Binary trees most commonly occur in the context of binary search trees, wherein keys are stored in a sorted fashion. However, there are many other applications of binary trees. Consider a set of resources organized as nodes in a binary tree. Processes need to lock resource nodes. A node may be locked if and only if none of its descendants and ancestors are locked. Your task is to design and implement an application programming interface (API) for locking.

A reasonable API is one with `isLock()`, `lock()`, and `unLock()` methods. Naïvely implemented the time complexity for these methods is $O(n)$, where n is the number of nodes. However these can be made to run in time $O(1)$, $O(h)$, and $O(h)$, respectively, where h is the height of the tree, if nodes have a parent field. Details are given in Solution 9.4 on Page 238.

Heaps

A heap is a data structure based on a binary tree. It efficiently implements an ADT called a priority queue. A priority queue resembles a queue, with one difference: each element has a "priority" associated with it, and deletion removes the element with the highest priority.

Suppose you are given a set of files, each containing stock trade information. Each trade appears as a separate line containing information about that trade. Lines begin with an integer-valued timestamp, and lines within a file are sorted in increasing order of timestamp. Suppose you were asked to design an algorithm that combines the set of files into a single file R in which trades are sorted by timestamp.

This problem can be solved by a multistage merge process, but there is a trivial solution based on a min-heap data structure. Entries are trade-file pairs and are ordered by the timestamp of the trade. Initially the min-heap contains the first trade from each file. Iteratively delete the minimum entry $e = (t, f)$ from the min-heap, write t to R, and add in the next entry in the file f. Details are given in Solution 10.1 on Page 248.

Hash tables

A hash table is a data structure used to store keys, optionally with corresponding values. Inserts, deletes and lookups run in $O(1)$ time on average. One caveat is that these operations require a good hash function—a mapping from the set of all possible keys to the integers which is similar to a uniform random assignment. Another is that if the number of keys that is to be stored is not known in advance then the hash table needs to be periodically resized, which depending on how the resizing is implemented, can lead to some updates having $\Theta(n)$ complexity.

Suppose you were asked to write an application that compares n programs for plagiarism. Specifically, your application is to break every program into overlapping character strings, each of length 100, and report on the number of strings that appear in each pair of programs. A hash table can be used to perform this check very efficiently if the right hash function is used. Details are given in Solution 12.13 on Page 286.

Binary search trees

Binary search trees (BSTs) are used to store objects that are comparable. The underlying idea is to organize the objects in a binary tree in which the nodes satisfy the BST property: the key stored at any node is greater than or equal to the keys stored in its

left subtree and less than or equal to the keys stored in its right subtree. Insertion and deletion can be implemented so that the height of the BST is $O(\log n)$, leading to fast ($O(\log n)$) lookup and update times. AVL trees and red-black trees are BST implementations that support this form of insertion and deletion.

BSTs are a workhorse of data structures and can be used to solve almost every data structures problem reasonably efficiently. It is common to augment the BST to make it possible to manipulate more complicated data, e.g., intervals, and efficiently support more complex queries, e.g., the number of elements in a range.

As an example application of BSTs, consider the following problem. You are given a set of line segments. Each segment is a closed interval $[l_i, r_i]$ of the x-axis, a color, and a height. For simplicity assume no two segments whose intervals overlap have the same height. When the x-axis is viewed from above the color at point x on the x-axis is the color of the highest segment that includes x. (If no segment contains x, the color is blank.) You are to implement a function that computes the sequence of colors as seen from the top.

The key idea is to sort the endpoints of the line segments and do a sweep from left-to-right. As we do the sweep, we maintain a list of line segments that intersect the current position as well as the highest line and its color. To quickly lookup the highest line in a set of intersecting lines we keep the current set in a BST, with the interval's height as its key. Details are given in Solution 14.20 on Page 327.

Other data structures

The data structures described above are the ones commonly used. Examples of other data structures that have more specialized applications include:

- Skip lists, which store a set of comparable items using a hierarchy of sorted linked lists. Lists higher in the hierarchy consist of increasingly smaller subsequences of the items. Skip lists implement the same functionality as balanced BSTs, but are simpler to code and faster, especially when used in a concurrent context.
- Treaps, which are a combination of a BST and a heap. When an element is inserted into a treap it is assigned a random key that is used in the heap organization. The advantage of a treap is that it is height-balanced with high probability and the insert and delete operations are considerably simpler than for deterministic height-balanced trees such as AVL and red-black trees.
- Fibonacci heaps, which consist of a series of trees. Insert, find minimum, decrease key, and merge (union) run in amortized constant time; delete and delete-minimum take $O(\log n)$ time. In particular Fibonacci heaps can be used to reduce the time complexity of Dijkstra's shortest path algorithm from $O((|E|+|V|)\log|V|)$ to $O(|E| + |V|\log|V|)$.
- Disjoint-set data structures, which are used to manipulate subsets. The basic operations are union (form the union of two subsets), and find (determine which set an element belongs to). These are used in a number of algorithms, notably in tracking connected components in an undirected graph and Kruskal's

algorithm for the minimum spanning tree. We use the disjoint-set data structure to solve the offline minimum problem (Solution 6.8 on Page 189).

- Tries, which are a tree-based data structure used to store strings. Unlike BSTs, nodes do not store keys; instead, the node's position in the tree determines the key it is associated with. Tries can have performance advantages with respect to BSTs and hash tables; they can also be used to solve the longest matching prefix problem (Solution 19.3 on Page 417).

Algorithm design patterns

An algorithm is a step-by-step procedure for performing a calculation. We classify common algorithm design patterns in Table 4.2. Roughly speaking, each pattern corresponds to a design methodology. An algorithm may use a combination of patterns.

Table 4.2: Algorithm design patterns.

Technique	Key points
Sorting	Uncover some structure by sorting the input.
Recursion	If the structure of the input is defined in a recursive manner, design a recursive algorithm that follows the input definition.
Divide and conquer	Divide the problem into two or more smaller independent subproblems and solve the original problem using solutions to the subproblems.
Dynamic programming	Compute solutions for smaller instances of a given problem and use these solutions to construct a solution to the problem. Cache for performance.
The greedy method	Compute a solution in stages, making choices that are locally optimum at step; these choices are never undone.
Incremental improvement	Quickly build a feasible solution and improve its quality with small, local updates.
Elimination	Identify and rule out potential solutions that are suboptimal or dominated by other solutions.
Parallelism	Decompose the problem into subproblems that can be solved independently on different machines.
Caching	Store computation and later look it up to save work.
Randomization	Use randomization within the algorithm to reduce complexity.
Approximation	Efficiently compute a suboptimum solution that is of acceptable quality.
State	Identify an appropriate notion of state.

Sorting

Certain problems become easier to understand, as well as solve, when the input is sorted. The solution to the calendar rendering problem (Problem 13.10 on Page 101) entails taking a set of intervals and computing the maximum number of intervals whose intersection is nonempty. Naïve strategies yield quadratic run times. However, once the interval endpoints have been sorted, it is easy to see that a point of maximum overlap can be determined by a linear time iteration through the endpoints.

Often it is not obvious what to sort on—for example, we could have sorted the intervals on starting points rather than endpoints. This sort sequence, which in some respects is more natural, does not work. However, some experimentation with it will likely lead to the correct criterion.

Sorting is not appropriate when an $O(n)$ (or better) algorithm is possible, e.g., determining the k-th largest element (Problem 11.13 on Page 89). Furthermore, sorting can obfuscate the problem. For example, given an array A of numbers, if we are to determine the maximum of $A[i] - A[j]$, for $i < j$, sorting destroys the order and complicates the problem.

Recursion

A recursive function consists of base cases, and calls to the same function with different arguments. A recursive algorithm is appropriate when the input is naturally expressed using recursive functions.

String matching exemplifies the use of recursion. Suppose you were asked to write a Boolean-valued function which takes a string and a matching expression, and returns `true` iff the matching expression "matches" the string. Specifically, the matching expression is itself a string, and could be

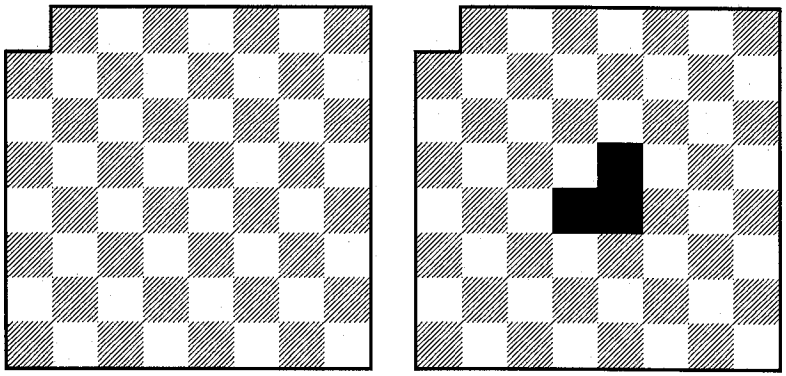
- x where x is a character, for simplicity assumed to be a lower-case letter (matches the string "x").
- . (matches any string of length 1).
- $x*$ (matches the string consisting of zero or more occurrences of the character x).
- .* (matches the string consisting of zero or more of any characters).
- $r_1 r_2$ where r_1 and r_2 are regular expressions of the given form (matches any string that is the concatenation of strings s_1 and s_2, where r_1 matches s_1 and r_2 matches s_2).

This problem can be solved by checking a number of cases based on the first one or two characters of the matching expression, and recursively matching the rest of the string. Details are given in Solution 6.23 on Page 206.

Divide and conquer

A divide and conquer algorithm works by decomposing a problem into two or more smaller independent subproblems, until it gets to instances that are simple enough to be solved directly; the results from the subproblems are then combined. More details and examples are given in Chapter 15; we illustrate the basic idea below.

A triomino is formed by joining three unit-sized squares in an L-shape. A mutilated chessboard (henceforth 8×8 Mboard) is made up of 64 unit-sized squares arranged in an 8×8 square, minus the top-left square, as depicted in Figure 4.2(a). Suppose you are asked to design an algorithm that computes a placement of 21 triominoes that covers the 8×8 Mboard. Since the 8×8 Mboard contains 63 squares, and we have 21 triominoes, a valid placement cannot have overlapping triominoes or triominoes which extend out of the 8×8 Mboard.



(a) An 8×8 Mboard. (b) Four 4×4 Mboards.

Figure 4.2: Mutilated chessboards.

Divide and conquer is a good strategy for this problem. Instead of the 8×8 Mboard, let's consider an $n \times n$ Mboard. A 2×2 Mboard can be covered with one triomino since it is of the same exact shape. You may hypothesize that a triomino placement for an $n \times n$ Mboard with the top-left square missing can be used to compute a placement for an $(n+1) \times (n+1)$ Mboard. However you will quickly see that this line of reasoning does not lead you anywhere.

Another hypothesis is that if a placement exists for an $n \times n$ Mboard, then one also exists for a $2n \times 2n$ Mboard. Now we can apply divide and conquer. work. Take four $n \times n$ Mboards and arrange them to form a $2n \times 2n$ square in such a way that three of the Mboards have their missing square set towards the center and one Mboard has its missing square outward to coincide with the missing corner of a $2n \times 2n$ Mboard, as shown in Figure 4.2(b). The gap in the center can be covered with a triomino and, by hypothesis, we can cover the four $n \times n$ Mboards with triominoes as well. Hence a placement exists for any n that is a power of 2. In particular, a placement exists for the $2^3 \times 2^3$ Mboard; the recursion used in the proof directly yields the placement.

Divide and conquer is usually implemented using recursion. However, the two concepts are not synonymous. Recursion is more general—subproblems do not have to be of the same form.

In addition to divide and conquer, we used the generalization principle above. The idea behind generalization is to find a problem that subsumes the given problem and is easier to solve. We used it to go from the 8×8 Mboard to the $2^n \times 2^n$ Mboard.

Other examples of divide and conquer include counting the number of pairs of elements in an array that are out of sorted order (Solution 15.2 on Page 334) and computing the closest pair of points in a set of points in the plane (Solution 15.3 on Page 335).

Dynamic programming

Dynamic programming (DP) is applicable when the problem has the "optimal substructure" property, that is, it is possible to reconstruct a solution to the given instance from solutions to subinstances of smaller problems of the same kind. A key aspect of DP is maintaining a cache of solutions to subinstances. DP can be implemented recursively (in which case the cache is typically a dynamic data structure such as a hash table or a BST), or iteratively (in which case the cache is usually a one- or multi-dimensional array). It is most natural to design a DP algorithm using recursion. Usually, but not always, it is more efficient to implement it using iteration.

As an example of the power of DP, consider the problem of determining the number of combinations of 2, 3, and 7 point plays that can generate a score of 222. Let $C(s)$ be the number of combinations that can generate a score of s. Then $C(222) = C(222-7) + C(222-3) + C(222-2)$, since a combination ending with a 2 point play is different from one ending with a 3 point play, and a combination ending with a 3 point play is different from one ending with a 7 point play, etc.

The recursion ends at small scores, specifically, when (1.) $s < 0 \Rightarrow C(s) = 0$, and (2.) $s = 0 \Rightarrow C(s) = 1$.

Implementing the recursion naïvely results in multiple calls to the same subinstance. Let $C(a) \rightarrow C(b)$ indicate that a call to C with input a directly calls C with input b. Then $C(213)$ will be called in the order $C(222) \rightarrow C(222-7) \rightarrow C((222-7)-2)$, as well as $C(222) \rightarrow C(222-3) \rightarrow C((222-3)-3) \rightarrow C(((222-3)-3)-3)$.

This phenomenon results in the run time increasing exponentially with the size of the input. The solution is to store previously computed values of C in an array of length 223. Details are given in Solution 15.15 on Page 354.

Sometimes it is profitable to study the set of partial solutions. Specifically it may be possible to "prune" dominated solutions, i.e., solutions which cannot be better than previously explored solutions. The candidate solutions are referred to as the "efficient frontier" that is propagated through the computation.

For example, if we are to implement a stack that supports a max operation, which returns the largest element stored in the stack, we can record for each element in the stack what the largest value stored at or below that element is by comparing the value of that element with the value of the largest element stored below it. Details are given in Solution 8.1 on Page 219. The largest rectangle under the skyline (Problem 15.8 on Page 120) provides a more sophisticated example of the efficient frontier concept.

Another consideration is how the partial solutions are organized. In the solution to the longest nondecreasing subsequence problem 15.6 on Page 340, it is better to keep the efficient frontier sorted by length of each subsequence rather than its final index.

The greedy method

A greedy algorithm is one which makes decisions that are locally optimum and never changes them. This strategy does not always yield the optimum solution. Furthermore, there may be multiple greedy algorithms for a given problem, and only some of them are optimum.

For example, consider $2n$ cities on a line, half of which are white, and the other half are black. We want to map white to black cities in a one-to-one fashion so that the total length of the road sections required to connect paired cities is minimized. Multiple pairs of cities may share a single section of road, e.g., if we have the pairing $(0, 4)$ and $(1, 2)$ then the section of road between Cities 0 and 4 can be used by Cities 1 and 2. The most straightforward greedy algorithm for this problem is to scan through the white cities, and, for each white city, pair it with the closest unpaired black city. It leads to suboptimum results: consider the case where white cities are at 0 and at 3 and black cities are at 2 and at 5. If the straightforward greedy algorithm processes the white city at 3 first, it pairs it with 2, forcing the cities at 0 and 5 to pair up, leading to a road length of 5, whereas the pairing of cities at 0 and 2, and 3 and 5 leads to a road length of 4.

However, a slightly more sophisticated greedy algorithm does lead to optimum results: iterate through all the cities in left-to-right order, pairing each city with the nearest unpaired city of opposite color. More succinctly, let W and B be the arrays of white and black city coordinates. Sort W and B, and pair $W[i]$ with $B[i]$. We can prove this leads to an optimum pairing by induction. The idea is that the pairing for the first city must be optimum, since if it were to be paired with any other city, we could always change its pairing to be with the nearest black city without adding any road.

Incremental improvement

When you are faced with the problem of computing an optimum solution, it is often straightforward to come up with a candidate solution, which may be a partial solution. This solution can be incrementally improved to make it optimum. This is especially true when a solution has to satisfy a set of constraints.

As an example consider a department with n graduate students and n professors. Each student begins with a rank ordered preference list of the professors based on how keen he is to work with each of them. Each professor has a similar preference list of students. Suppose you were asked to devise an algorithm which takes as input the preference lists and outputs a one-to-one pairing of students and advisers in which there are no student-adviser pairs $(s0, a0)$ and $(s1, a1)$ such that $s0$ prefers $a1$ to $a0$ and $a1$ prefers $s0$ to $s1$.

Here is an algorithm for this problem in the spirit of incremental improvement. Each student who does not have an adviser "proposes" to the most-preferred professor to whom he has not yet proposed. Each professor then considers all the students who have proposed to him and says to the student in this set he most prefers "I accept you"; he says "no" to the rest. The professor is then provisionally matched to

a student; this is the candidate solution. In each subsequent round, each student who does not have an adviser proposes to the professor to whom he has not yet proposed who is highest on his preference list. He does this regardless of whether the professor has already been matched with a student. The professor once again replies with a single accept, rejecting the rest. In particular, he may leave a student with whom he is currently paired. That this algorithm is correct is nontrivial—details are presented in Solution 21.18 on Page 459.

Many other algorithms are in this spirit: the standard algorithms for bipartite matching (Solution 21.19 on Page 460), maximum flow (Solution 21.21 on Page 462), and computing all pairs of shortest paths in a graph (Solutions 16.11 on Page 386 and 16.12 on Page 387) use incremental improvement. Other famous examples include the simplex algorithm for linear programming, and Euler's algorithm for computing a path in a graph which covers each edge once.

Sometimes it is easier to start with an infeasible solution that has a lower cost than the optimum solution, and incrementally update it to arrive at a feasible solution that is optimum. The standard algorithms for computing an MST (Solution 17.6 on Page 393) and shortest paths in a graph from a designated vertex (Solution 16.9 on Page 384) proceed in this fashion.

It is noteworthy that naïvely applying incremental improvement does not always work. For the professor-student pairing example above, if we begin with an arbitrary pairing of professors and students, and search for pairs p and s such that p prefers s to his current student, and s prefers p to his current professor and reassign such pairs, the procedure will not always converge.

Incremental improvement is often useful when designing heuristics, i.e., algorithms which are usually faster and/or simpler to implement than algorithms which compute an optimum result, but may return a suboptimal result. The algorithm we present for computing a tour for a traveling salesman (Solution 17.6 on Page 393) is in this spirit.

Elimination

One common approach to designing an efficient algorithm is to use elimination—that is to identify and rule out potential solutions that are suboptimal or dominated by other solutions. Binary search, which is the subject of a number of problems in Chapter 11, uses elimination. Solution 11.9 on Page 267, where we use elimination to compute the square root of a real number, is especially instructive. Below we consider a fairly sophisticated application of elimination.

Suppose you have to build a distributed storage system. A large number, n, of users will share data on your system, which consists of m servers, numbered from 0 to $m-1$. One way to distribute users across servers is to assign the user with login ID l to the server $h(l) \bmod m$, where $h()$ is a hash function. If the hash function does a good job, this approach distributes users uniformly across servers. However, if certain users require much more storage than others, some servers may be overloaded while others idle.

Let b_i be the number of bytes of storage required by user i. We will use values $k_0 < k_1 < \cdots < k_{m-2}$ to partition users across the m servers—a user with hash code c gets assigned to the server with the lowest ID i such that $c \leq k_i$, or to server $m-1$ if no such i exists. We would like to select $k_0, k_1, \ldots, k_{m-2}$ to minimize the maximum number of bytes stored at any server.

The optimum values for $k_0, k_1, \ldots, k_{m-2}$ can be computed via DP—the essence of the program is to add one server at a time. The straightforward formulation has an $O(nm^2)$ time complexity.

However, there is a much faster approach based on elimination. The search for values $k_0, k_1, \ldots, k_{m-2}$ such that no server stores more than b bytes can be performed in $O(n)$ time by greedily selecting values for the k_is. We can then perform binary search on b to get the minimum b and the corresponding values for $k_0, k_1, \ldots, k_{m-2}$. The resulting time complexity is $O(n \log W)$, where $W = \sum_{i=0}^{m-1} b_i$.

For the case of 10000 users and 100 servers, the DP algorithm took over an hour; the approach using binary search for b with greedy assignment took 0.1 seconds. Details are given in Solution 15.24 on Page 365.

The takeaway is that there may be qualitatively different ways to search for a solution, and that it is important to look for ways in which to eliminate candidates. The efficient frontier concept, described on Page 31, has some commonalities with elimination.

Parallelism

In the context of interview questions, parallelism is useful when dealing with scale, i.e., when the problem is too large to fit on a single machine or would take an unacceptably long time on a single machine. The key insight you need to display is that you know how to decompose the problem so that

1. each subproblem can be solved relatively independently, and
2. the solution to the original problem can be efficiently constructed from solutions to the subproblems.

Efficiency is typically measured in terms of central processing unit (CPU) time, random access memory (RAM), network bandwidth, number of memory and database accesses, etc.

Consider the problem of sorting a petascale integer array. If we know the distribution of the numbers, the best approach would be to define equal-sized ranges of integers and send one range to one machine for sorting. The sorted numbers would just need to be concatenated in the correct order. If the distribution is not known then we can send equal-sized arbitrary subsets to each machine and then merge the sorted results, e.g., using a min-heap. Details are given in Solution 18.14 on Page 414.

Caching

Caching is a great tool whenever computations are repeated. For example, the central idea behind dynamic programming is caching results from intermediate computations. Caching is also extremely useful when implementing a service that is expected

to respond to many requests over time, and many requests are repeated. Workloads on web services exhibit this property. Solution 18.1 on Page 403 sketches the design of an online spell correction service; one of the key issues is performing cache updates in the presence of concurrent requests.

Randomization

Suppose you were asked to write a routine that takes an array A of n elements and an integer k between 1 and n, and returns the k-th largest element in A.

This problem can be solved by first sorting the array, and returning the element at index k in the sorted array. The time complexity of this approach is $O(n \log n)$. However, sorting performs far more work than is needed. A better approach is to eliminate parts of the array. We could use the median to determine the $n/2$ largest elements of A; if $n/2 \geq k$, the desired element is in this set, otherwise we search for the $(k - n/2)$-th largest element in the $n/2$ smallest elements.

It is possible, though nontrivial, to compute the median in $O(n)$ time without using randomization. However, an approach that works well is to select an index r at random and reorder the array so that elements greater than or equal to $A[r]$ appear first, followed by $A[r]$, followed by elements less than or equal to $A[r]$. Let $A[r]$ be the k-th element in the reordered array A'. If $k = n/2$, $A'[k] = A[r]$ is the desired element. If $k > n/2$, we search for the $n/2$-th largest element in $A'[0 : k-1]$. Otherwise we search for the $(n/2-k)$-th largest element in $A'[k+1 : n-1]$. The closer $A[r]$ is the true median, the faster the algorithm runs. A formal analysis shows that the probability of the randomly selected element repeatedly being far from the desired element falls off exponentially with n. Details are given in Solution 11.13 on Page 270.

Randomization can also be used to create "signatures" to reduce the complexity of search, analogous to the use of hash functions. Consider the problem of determining whether an $m \times m$ array S of integers is a subarray of an $n \times n$ array T. Formally, we say S is a subarray of T iff there are p, q such that $S[i][j] = T[p+i][q+j]$, for all $0 \leq i, j \leq m-1$. The brute-force approach to checking if S is a subarray of T has complexity $O(n^2m^2)$—$O(n^2)$ individual checks, each of complexity $O(m^2)$. We can improve the complexity to $O(n^2m)$ by computing a hash code for S and then computing the hash codes for $m \times m$ subarrays of T. The latter hash codes can be computed incrementally in $O(m)$ time if the hash function is chosen appropriately. For example, if the hash code is simply the XOR of all the elements of the subarray, the hash code for a subarray shifted over by one column can be computed by XORing the new elements and the removed elements with the previous hash code. A similar approach works for more complex hash functions, specifically for those that are a polynomial.

Approximation

In the real-world it is routine to be given a problem that is difficult to solve exactly, either because of its intrinsic complexity, or the complexity of the code required. Developers need to recognize such problems, and be ready to discuss alternatives

with the author of the problem. In practice a solution that is "close" to the optimum solution is usually perfectly acceptable.

Let $\{A_0, A_1, \ldots, A_{n-1}\}$ be a set of n cities, as in Figure 4.3. Suppose we need to choose a subset of A to locate warehouses. Specifically, we want to choose k cities in such a way that cities are close to the warehouses. Define the cost of a warehouse assignment to be the maximum distance of any city to a warehouse.

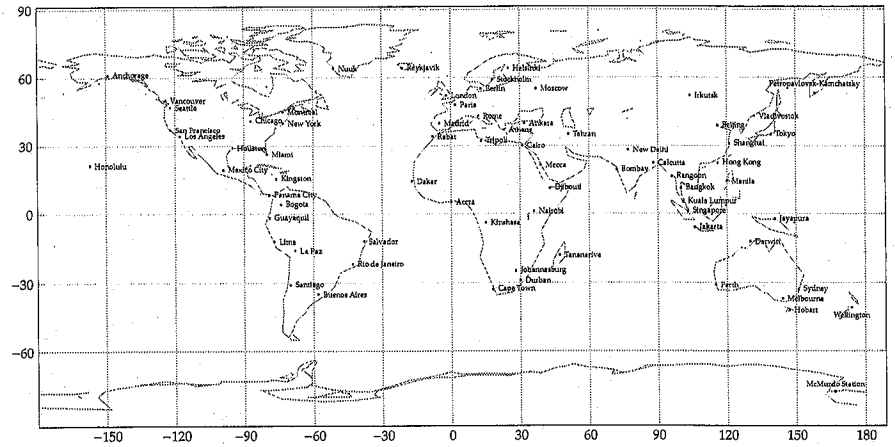


Figure 4.3: An instance of the warehouse location problem. The distance between cities at (p, q) and (r, s) is $\sqrt{(p-r)^2 + (q-s)^2}$.

The problem of finding a warehouse assignment that has the minimum cost is known to be NP-complete. However, consider the following algorithm for computing k cities. We pick the first warehouse to be the city for which the cost is minimized—this takes $\Theta(n^2)$ time since we try each city one at a time and check its distance to every other city. Now let's say we have selected the first $i-1$ warehouses $\{c_1, c_2, \ldots, c_{i-1}\}$ and are trying to choose the i-th warehouse. A reasonable choice for c_i is the city that is the farthest from the $i-1$ warehouses already chosen. This city can be computed in $O(ni)$ time. This greedy algorithm yields a solution whose cost is no more than $2\times$ that of the optimum solution; some heuristic tweaks can be used to further improve the quality. Details are given in Solution 17.7 on Page 394.

As another example of approximation, consider the problem of determining the k most frequent elements of a very large array. The direct approach of maintaining counts for each element may not be feasible because of space constraints. A natural approach is to *sample* the set to determine a set of candidates, exact counts for which are then determined in a second pass. The size of the candidate set depends on the distribution of the elements.

STATE

Formally, the state of a system is information that is sufficient to determine how that system evolves as a function of future inputs. Identifying the right notion of state can be critical to coming up with an algorithm that is time and space efficient, as well as easy to implement and prove correct.

There may be multiple ways in which state can be defined, all of which lead to correct algorithms. When computing the max-difference (Problem 6.3 on Page 53), we could use the values of the elements at all prior indices as the state when we iterate through the array. Of course, this is inefficient, since all we really need is the minimum value.

One solution to computing the Levenshtein distance between two strings (Problem 15.11 on Page 120) entails creating a 2D array whose dimensions are $(m+1)\times(n+1)$, where m and n are the lengths of the strings being compared. For large strings this size may be unacceptably large. The algorithm iteratively fills rows of the array, and reads values from the current row and the previous row. This observation can be used to reduce the memory needed to two rows. A more careful implementation can reduce the memory required to just one row.

More generally, the efficient frontier concept on Page 31 demonstrates how an algorithm can be made to run faster and with less memory if state is chosen carefully. Other examples illustrating the benefits of careful state selection include string matching (Problem 6.20 on Page 59) and lazy initialization (Problem 6.2 on Page 53).

Abstract analysis patterns

The mathematician George Polya wrote a book *How to Solve It* that describes a number of heuristics for problem solving. Inspired by this work we present some heuristics that are effective on common interview problems; they are summarized in Table 4.3 on the following page.

CASE ANALYSIS

In case analysis a problem is divided into a number of separate cases, and analyzing each such case individually suffices to solve the initial problem. Cases do not have to be mutually exclusive; however, they must be exhaustive, that is cover all possibilities. For example, to prove that for all n, $n^3 \bmod 3$ is 0, 1, or 8, we can consider the cases $n = 3m$, $n = 3m + 1$, and $n = 3m + 2$. These cases are individually easy to prove, and are exhaustive. Case analysis is commonly used in mathematics and games of strategy. Here we consider an application of case analysis to algorithm design.

Suppose you are given a set S of 25 distinct integers and a CPU that has a special instruction, `SORT5`, that can sort five integers in one cycle. Your task is to identify the largest, second-largest, and third-largest integers in S using `SORT5` to compare and sort subsets of S; furthermore, you must minimize the number of calls to `SORT5`.

If all we had to compute was the largest integer in the set, the optimum approach would be to form five disjoint subsets $S_1, \ldots, S_5$ of S, sort each subset, and then sort

Table 4.3: Abstract analysis techniques.

Analysis principle	Key points
Case analysis	Split the input/execution into a number of cases and solve each case in isolation.
Small examples	Find a solution to small concrete instances of the problem and then build a solution that can be generalized to arbitrary instances.
Iterative refinement	Most problems can be solved using a brute-force approach. Find such a solution and improve upon it.
Reduction	Use a well known solution to some other problem as a subroutine.
Graph modeling	Describe the problem using a graph and solve it using an existing algorithm.
Write an equation	Express relationships in the problem in the form of equations (or inequalities).
Variation	Solve a slightly different (possibly more general) problem and map its solution to the given problem.
Invariants	Find a function of the state of the given system that remains constant in the presence of (possibly restricted) updates to the state. Use this function to design an algorithm, prove correctness, or show an impossibility result.

$\{\max S_1, \ldots, \max S_5\}$. This takes six calls to SORT5 but leaves ambiguity about the second and third largest integers.

It may seem like many additional calls to SORT5 are still needed. However if you do a careful case analysis and eliminate all $x \in S$ for which there are at least three integers in S larger than x, only five integers remain and hence just one more call to SORT5 is needed to compute the result. Details are given in Solution 21.2 on Page 447.

Small examples

Problems that seem difficult to solve in the abstract can become much more tractable when you examine small concrete instances. For instance, consider the following problem. Five hundred closed doors along a corridor are numbered from 1 to 500. A person walks through the corridor and opens each door. Another person walks through the corridor and closes every alternate door. Continuing in this manner, the i-th person comes and toggles the state (open or closed) of every i-th door starting from Door i. You must determine exactly how many doors are open after the 500-th person has walked through the corridor.

It is difficult to solve this problem using an abstract approach, e.g., introducing Boolean variables for the state of each door and a state update function. However if you try the same problem with $1, 2, 3, 4, 10$, and 20 doors, it takes a short time to see that the doors that remain open are $1, 4, 9, 16, \ldots$, regardless of the total number of doors. The 10 doors case is illustrated in Figure 4.4 on the facing page. Now

the pattern is obvious—the doors that remain open are those corresponding to the perfect squares. Once you make this connection, it is easy to prove it for the general case. Hence the total number of open doors is $\lfloor \sqrt{500} \rfloor = 22$. Solution 21.1 on Page 446 develops this analysis in more detail.

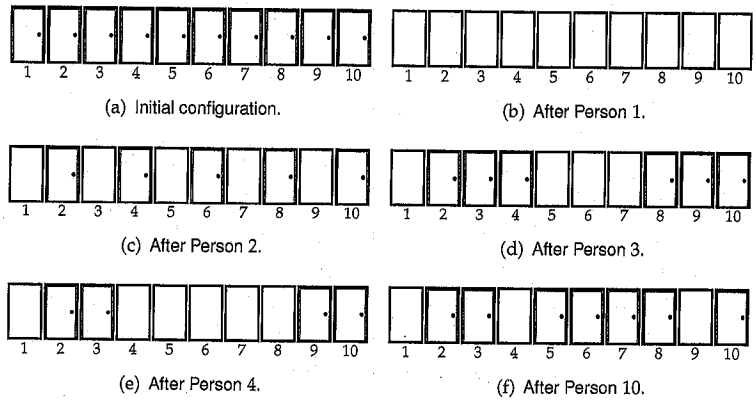


Figure 4.4: Progressive updates to 10 doors.

Optimally selecting a red card (Problem 20.18 on Page 160) and avoiding losing at the alternating coin pickup game (Problem 21.20 on Page 168) are other problems that benefit from use of the "small example" principle.

Iterative refinement of a brute-force solution

Many problems can be solved optimally by a simple algorithm that has a high time/space complexity—this is sometimes referred to as a brute-force solution. Other terms are *exhaustive search* and *generate-and-test*. Often this algorithm can be refined to one that is faster. At the very least it may offer hints into the nature of the problem.

As an example, suppose you were asked to write a function that takes an array A of n numbers, and rearranges A's elements to get a new array B having the property that $B[0] \leq B[1] \geq B[2] \leq B[3] \geq B[4] \leq B[5] \geq \cdots$.

One straightforward solution is to sort A and interleave the bottom and top halves of the sorted array. Alternately, we could sort A and then swap the elements at the pairs $(A[1], A[2]), (A[3], A[4]), \ldots$. Both these approaches have the same time complexity as sorting, namely $O(n \log n)$.

You will soon realize that it is not necessary to sort A to achieve the desired configuration—you could simply rearrange the elements around the median, and then perform the interleaving. Median finding can be performed in time $O(n)$ (Solution 11.13 on Page 270), which is the overall time complexity of this approach.

Finally, you may notice that the desired ordering is very local, and realize that it is not necessary to find the median. Iterating through the array and swapping $A[i]$ and $A[i+1]$ when i is even and $A[i] > A[i+1]$ or i is odd and $A[i] < A[i+1]$ achieves the desired configuration. In code:

```
template <typename T>
void rearrange(vector<T> &A) {
  for (int i = 0; i < A.size() - 1; ++i) {
    if ((i & 1 && A[i] < A[i + 1]) || ((i & 1) == 0 && A[i] > A[i + 1])) {
      swap(A[i], A[i + 1]);
    }
  }
}
```

This approach has time complexity $O(n)$, which is the same as the approach based on median finding. However it is much easier to implement, and operates in an online fashion, i.e., it never needs to store more than two elements in memory or read a previous element.

As another example of iterative refinement, consider the problem of string search (Problem 6.20 on Page 59): given two strings s (search string) and t (text), find all occurrences of s in t. Since s can occur at any offset in t, the brute-force solution is to test for a match at every offset. This algorithm is perfectly correct; its time complexity is $O(nm)$, where n and m are the lengths of s and t.

After trying some examples, you may see that there are several ways to improve the time complexity of the brute-force algorithm. As an example, if the character $t[i]$ is not present in s you can advance the matching by n characters. Furthermore, this skipping works better if we match the search string from its end and work backwards. These refinements will make the algorithm very fast (linear time) on random text and search strings; however, the worst-case complexity remains $O(nm)$.

You can make the additional observation that a partial match of s that does not result in a full match implies other offsets that cannot lead to full matches. If s = *abdabcabc* and if, starting backwards, we have a partial match up to *abcabc* that does not result in a full match, we know that the next possible matching offset has to be at least three positions ahead (where we can match the second *abc* from the partial match).

By putting together these refinements you will have arrived at the famous Boyer-Moore string search algorithm—its worst-case time complexity is $O(n + m)$ (which is the best possible from a theoretical perspective); it is also one of the fastest string search algorithms in practice.

Many other sophisticated algorithms can be developed in this fashion. As another example, the brute-force solution to computing the maximum subarray sum for an integer array of length n is to compute the sum of all subarrays, which has $O(n^3)$ time complexity. This can be improved to $O(n^2)$ by precomputing the sums of all the prefixes of the given arrays; this allows the sum of a subarray to be computed in $O(1)$ time. The natural divide and conquer algorithm has an $O(n \log n)$ time complexity. Finally, one can observe that a maximum subarray must end at one of n indices, and the maximum subarray sum for a subarray ending at index i can be computed from previous maximum subarray sums, which leads to an $O(n)$ algorithm. Details are presented on Page 117.

Reduction

Consider the problem of finding if one string is a rotation of the other, e.g., "car" and "arc" are rotations of each other. A natural approach may be to rotate the first string by every possible offset and then compare it with the second string. This algorithm would have quadratic time complexity.

You may notice that this problem is quite similar to string search, which can be done in linear time, albeit using a somewhat complex algorithm. Therefore it is natural to try to reduce this problem to string search. Indeed, if we concatenate the second string with itself and search for the first string in the resulting string, we will find a match iff the two original strings are rotations of each other. This reduction yields a linear time algorithm for our problem.

The reduction principle is also illustrated in the problem of checking whether a road network is resilient in the presence of blockages (Problem 16.4 on Page 133) and the problem of finding the minimum number of pictures needed to photograph a set of teams (Problem 16.7 on Page 135).

Usually you try to reduce the given problem to an easier problem. Sometimes, however, you need to reduce a problem known to be difficult to the given problem. This shows that the given problem is difficult, which justifies heuristics and approximate solutions. Such scenarios are described in more detail in Chapter 17.

Graph modeling

Drawing pictures is a great way to brainstorm for a potential solution. If the relationships in a given problem can be represented using a graph, quite often the problem can be reduced to a well-known graph problem. For example, suppose you are given a set of exchange rates among currencies and you want to determine if an arbitrage exists, i.e., there is a way by which you can start with one unit of some currency C and perform a series of barters which results in having more than one unit of C.

Table 4.4 shows a representative example. An arbitrage is possible for this set of exchange rates: $1\text{ USD} \rightarrow 1 \times 0.8123 = 0.8123\text{ EUR} \rightarrow 0.8123 \times 1.2010 = 0.9755723\text{ CHF} \rightarrow 0.9755723 \times 80.39 = 78.426257197\text{ JPY} \rightarrow 78.426257197 \times 0.0128 = 1.00385609212\text{ USD}$.

Table 4.4: Exchange rates for seven major currencies.

Symbol	USD	EUR	GBP	JPY	CHF	CAD	AUD
USD	1	0.8148	0.6404	78.125	0.9784	0.9924	0.9465
EUR	1.2275	1	0.7860	96.55	1.2010	1.2182	1.1616
GBP	1.5617	1.2724	1	122.83	1.5280	1.5498	1.4778
JPY	0.0128	0.0104	0.0081	1	1.2442	0.0126	0.0120
CHF	1.0219	0.8327	0.6546	80.39	1	1.0142	0.9672
CAD	1.0076	0.8206	0.6453	79.26	0.9859	1	0.9535
AUD	1.0567	0.8609	0.6767	83.12	1.0339	1.0487	1

We can model the problem with a graph where currencies correspond to vertices, exchanges correspond to edges, and the edge weight is set to the logarithm of the

exchange rate. If we can find a cycle in the graph with a positive weight, we would have found such a series of exchanges. Such a cycle can be solved using the Bellman-Ford algorithm, as described in Solution 16.12 on Page 387.

Write an equation

Some problems can be solved by expressing them in the language of mathematics. Suppose you were asked to write an algorithm that computes binomial coefficients, $\binom{n}{k} = \frac{n!}{k!(n-k)!}$.

The problem with computing the binomial coefficient directly from the definition is that the factorial function grows quickly and can overflow an integer variable. If we use floating point representations for numbers, we lose precision and the problem of overflow does not go away. These problems potentially exist even if the final value of $\binom{n}{k}$ is small. One can try to factor the numerator and denominator and try to cancel out common terms but factorization is itself a hard problem.

The binomial coefficients satisfy the *addition formula*:

$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$$

This identity leads to a straightforward recursion for computing $\binom{n}{k}$ which avoids the problems described above. DP has to be used to achieve good time complexity—details are in Solution 15.14 on Page 353.

Variation

The idea of the variation pattern is to solve a slightly different (possibly more general) problem and map its solution to your problem.

Suppose we were asked to design an algorithm which takes as input an undirected graph and produces as output a black or white coloring of the vertices such that for every vertex at least half of its neighbors differ in color from it.

We could try to solve this problem by assigning arbitrary colors to vertices and then flipping colors wherever constraints are not met. However this approach may lead to increasing the number of vertices that do not satisfy the constraint.

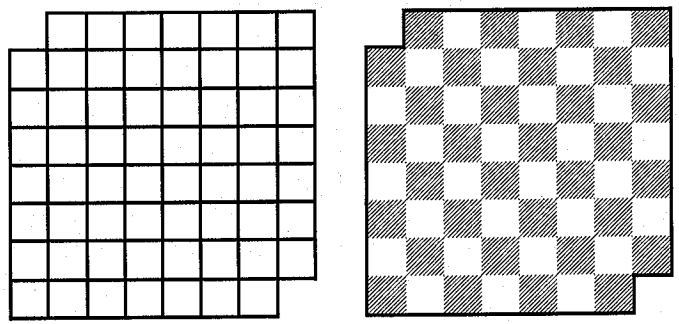
It turns out we can define a slightly different problem whose solution will yield the desired coloring. Define an edge to be *diverse* if its ends have different colors. It is straightforward to verify that a coloring that maximizes the number of diverse edges also satisfies the constraint of the original problem, so there always exists a coloring satisfying the constraint.

It is not necessary to find a coloring that maximizes the number of diverse edges. All that is needed is a coloring in which the set of diverse edges is maximal with respect to single vertex flips. Such a coloring can be computed efficiently; details are given in Problem 15.29 on Page 128.

INVARIANTS

The following problem was popular at interviews in the early 1990s. You are given an 8×8 square with two unit sized squares at the opposite ends of a diagonal removed, leaving 62 squares, as illustrated in Figure 4.5(a). You are given 31 rectangular dominoes. Each can cover exactly two squares. How would you cover all the 62 squares with the dominoes?

You can spend hours trying unsuccessfully to find such a covering. This experience will teach you that a problem may be intentionally worded to mislead you into following a futile path.



(a) An 8×8 square, minus two unit squares at opposite corners.

(b) A chessboard, with two diagonally opposite corners removed.

Figure 4.5: Invariant analysis exploiting auxiliary elements.

Here is a simple argument that no covering exists. Think of the 8×8 square as a chessboard as shown in Figure 4.5(b). Then the two removed squares will always have the same color, so there will be either 30 black and 32 white squares to be covered, or 32 black and 30 white squares to be covered. Each domino will cover one black and one white square, so the number of black and white squares covered as you successively put down the dominoes is equal. Hence it is impossible to cover the given chessboard.

This proof of impossibility is an example of invariant analysis. An invariant is a function of the state of a system being analyzed that remains constant in the presence of (possibly restricted) updates to the state. Invariant analysis is particularly powerful at proving impossibility results as we just saw with the chessboard tiling problem. The challenge is finding a simple invariant.

The argument above also used the auxiliary elements pattern, in which we added a new element to our problem to get closer to a solution. The original problem did not talk about the colors of individual squares; adding these colors made proving impossibility much easier.

It is possible to prove impossibility without appealing to square colors. Specifically, orient the board with the missing pieces on the lower right and upper left. An impossibility proof exists based on a case-analysis for each column on the height of

the highest domino that is parallel to the base. However, the proof given above is much simpler.

Invariant analysis can be used to design algorithms, as well as prove impossibility results. In the coin selection problem, sixteen coins are arranged in a line, as in Figure 4.6. Two players, F and S, take turns at choosing one coin each—they can only choose from the two coins at the ends of the line. Player F goes first. The game ends when all the coins have been picked up. The player whose coins have the higher total value wins.

Figure 4.6: Coins in a row.

The optimum strategy for F can be computed using DP (Solution 15.18 on Page 357). However, if F's goal is simply to ensure he does not do worse than S, he can achieve this goal with much less computation. Specifically, he can number the coins from 1 to 16 from left-to-right, and compute the sum of the even-index coins and the sum of the odd-index coins. Suppose the odd-index sum is larger. Then F can force S to always select an even-index coin by selecting the odd-index coins when it is his own turn, ensuring that S cannot win. The same principle holds when the even-index sum is larger, or the sums are equal. Details are given in Solution 21.5 on Page 447.

Invariant analysis can be used with symmetry to solve very difficult problems, sometimes in less than intuitive ways. This is illustrated by the game known as "chomp" in which Player F and Player S alternately take bites from a chocolate bar. The chocolate bar is an $n \times n$ rectangle; a bite must remove a square and all squares above and to the right in the chocolate bar. The first player to eat the lower leftmost square, which is poisoned, loses. Player F can force a win by first selecting the square immediately above and to the right of the poisoned square, leaving the bar shaped like an L, with equal vertical and horizontal sides. Now whatever move S makes, F can play a symmetric move about the line bisecting the chocolate bar through the poisoned square to recreate the L shape (this is the invariant), which forces S to be the first to consume the poisoned square. Details are given in Solution 21.6 on Page 448.

Algorithm design using invariants is also illustrated in Solution 12.8 on Page 281 (permute the characters in a string to form a palindrome) and in Solution 13.14 on Page 303 (find three elements in an array that sum to a given number).

Complexity Analysis

The run time of an algorithm depends on the size of its input. One common approach to capture the run time dependency is by expressing asymptotic bounds on the worst-case run time as a function of the input size. Specifically, the run time of an algorithm on an input of size n is $O(f(n))$ if, for sufficiently large n, the run time is not more than $f(n)$ times a constant. The big-O notation simply indicates an upper bound;

if the run time is asymptotically proportional to $f(n)$, the complexity is written as $\Theta(f(n))$. (Note that the big-O notation is widely used where sometimes Θ is more appropriate.) The notation $\Omega(f(n))$ is used to denote an asymptotic lower bound of $f(n)$ on the time complexity of an algorithm.

As an example, searching an unsorted array of integers of length n, for a given integer, has an asymptotic complexity of $\Theta(n)$ since in the worst-case, the given integer may not be present. Similarly, consider the naïve algorithm for testing primality that tries all numbers from 2 to the square root of the input number n. What is its complexity? In the best case, n is divisible by 2. However in the worst-case the input may be a prime, so the algorithm performs $\sqrt{n}$ iterations. Furthermore, since the number n requires $\lg n$ bits to encode, this algorithm's complexity is actually exponential in the size of the input. The big-Omega notation is illustrated by the $\Omega(n \log n)$ lower bound on any comparison-based array sorting algorithm.

Generally speaking, if an algorithm has a run time that is a polynomial, i.e., $O(n^k)$ for some fixed k, where n is the size of the input, it is considered to be efficient; otherwise it is inefficient. Notable exceptions exist—for example, the simplex algorithm for linear programming is not polynomial but works very well in practice. On the other hand, the AKS primality testing algorithm has polynomial run time but the degree of the polynomial is too high for it to be competitive with randomized algorithms for primality testing.

Complexity theory is applied as a similar way when analyzing the space requirements of an algorithm. Usually, the space needed to read in an instance is not included; otherwise, every algorithm would have $\Omega(n)$ space complexity.

Several of our problems call for an algorithm that uses $O(1)$ space. Conceptually, the memory used by such an algorithm should not depend on the size of the input instance. Specifically, it should be possible to implement the algorithm without dynamic memory allocation (explicitly, or indirectly, e.g., through library routines). Furthermore, the maximum depth of the function call stack should also be a constant, independent of the input. The standard algorithm for depth-first search of a graph is an example of an algorithm that does not perform any dynamic allocation, but uses the function call stack for implicit storage—its space complexity is not $O(1)$.

A streaming algorithm is one in which the input is presented as a sequence of items and is examined in only a few passes (typically just one). These algorithms have limited memory available to them (much less than the input size) and also limited processing time per item. Algorithms for computing summary statistics on log file data often fall into this category.

As a rule, algorithms should be designed with the goal of reducing the worst-case complexity rather than average-case complexity for several reasons:

1. It is very difficult to define meaningful distributions on the inputs.
2. Pathological inputs are more likely than statistical models may predict. A worst-case input for a naïve implementation of quicksort is one where all entries are the same, which is not unlikely in a practical setting.
3. Malicious users may exploit bad worst-case performance to create denial-of-service attacks.

Part II

Problems

CHAPTER

5

Primitive Types

Representation is the essence of programming.

— *"The Mythical Man Month,"*
F. P. BROOKS, 1975

A program updates variables in memory according to the instructions in the program. The variables are classified according to their type—a type is a classification of data that spells out possible values for that type and the operations that can be done on that type.

Types can be primitive, i.e., provided by the language, or defined by the programmer. The set of primitive types depends on the language. For example, the primitive types in C++ are `bool`, `char`, `short`, `int`, `long`, `float`, and `double`, and in Java are `boolean`, `char`, `byte`, `short`, `int`, `long`, `float`, and `double`. A programmer can define a complex number type as a pair of doubles, one for the real and one for the imaginary part.

Problems involving manipulation of bit-level data are often asked in interviews. An old question goes as follows. Given two integer-valued variables a and b, the straightforward way of swapping their contents is to use a temporary variable—`temp = a; a = b; b = temp;`. The question is: can you swap without using an additional variable? Surprisingly it is possible—`a = a ^ b; b = a ^ b; a = a ^ b;`, where `^` is the binary bitwise-XOR operator, does the trick. The same code can be expressed more tersely as `a ^= b ^= a ^= b;`.

It is easy to introduce errors in code that manipulates bit-level data—when you play with bits, expect to get bitten.

5.1 COMPUTING PARITY

The parity of a sequence of bits is 1 if the number of 1s in the sequence is odd; otherwise, it is 0. Parity checks are used to detect single bit errors in data storage and communication. It is fairly straightforward to write code that computes the parity of a single 64-bit nonnegative integer.

Problem 5.1: How would you go about computing the parity of a very large number of 64-bit nonnegative integers? *pg. 173*

5.2 SWAP BITS

There are a number of ways in which bit manipulations can be accelerated. For example, the expression $x \,\&\, (x-1)$ equals x with the least significant bit cleared;

$x \,\&\, !(x-1)$ extracts the lowest set bit of x (all other bits are cleared); and $x \oplus (x \gg 1)$ is the standard (binary-reflected) Gray code for x.

Problem 5.2: A 64-bit integer can be viewed as an array of 64 bits, with the bit at index 0 corresponding to the least significant bit, and the bit at index 63 corresponding to the most significant bit. Implement code that takes as input a 64-bit integer x and swaps the bits at indices i and j. *pg. 174*

5.3 Bit reversal

Here is a bit fiddling problem that is concerned with restructuring.

Problem 5.3: Write a function that takes a 64-bit integer x and returns a 64-bit integer consisting of the bits of x in reverse order. *pg. 174*

5.4 Closest integers with the same weight

Define the number of bits that are set to 1 in an unsigned 64-bit integer x to be the *weight* of x. Let S_k denote the set of unsigned 64-bit integers whose weight is k.

Problem 5.4: Suppose $x \in S_k$, and k is not 0 or 64. How would you compute $y \in S_k \setminus \{x\}$ such that $|y - x|$ is minimum? *pg. 174*

5.5 The power set

The power set of a set S is the set of all subsets of S, including both the empty set $\emptyset$ and S itself. The power set of $\{A, B, C\}$ is graphically illustrated in Figure 5.1.

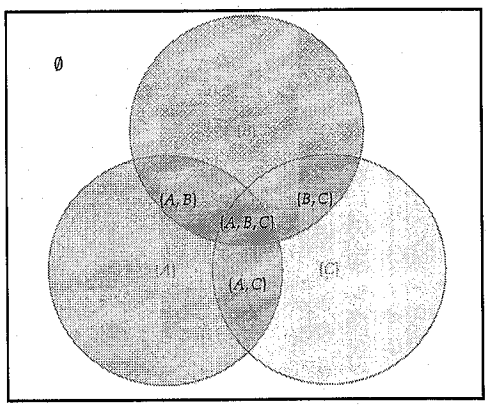


Figure 5.1: The power set of $\{A, B, C\}$ is $\{\emptyset, \{A\}, \{B\}, \{C\}, \{A, B\}, \{B, C\}, \{A, C\}, \{A, B, C\}\}$.

Problem 5.5: Implement a method that takes as input a set S of distinct elements, and prints the power set of S. Print the subsets one per line, with elements separated by commas. *pg. 175*

5.6 String and integer conversions

A string is a sequence of characters. A string may encode an integer, e.g., "123" encodes 123. In this problem, you are to implement methods that take a string representing an integer and return the corresponding integer, and vice versa.

Your code should handle negative integers. It should throw an exception if the string does not encode an integer, e.g., "123abc" is not a valid encoding.

Languages such as C++ and Java have library functions for performing this conversion—`stoi` in C++ and `parseInt` in Java go from strings to integers; `to_string` in C++ and `toString` in Java go from integers to strings. You cannot use these functions. (Imagine you are implementing the corresponding library.)

Problem 5.6: Implement string/integer inter-conversion functions. Use the following function signatures: `String intToString(int x)` and `int stringToInt(String s)`. *pg. 176*

5.7 Base conversion

In the decimal system, the position of a digit is used to signify the power of 10 that digit is to be multiplied with. For example, "314" denotes the number $3 \times 100 + 1 \times 10 + 4 \times 1$. (Note that zero, which is not needed in other systems, is essential in the decimal system, since a zero can be used to skip a power.)

The decimal system is an example of a positional number system, wherein the same symbol is used for different orders of magnitude (for example, the "ones place", "tens place", "hundreds place"). This system greatly simplified arithmetic and led to its widespread adoption.

The base b number system generalizes the above: the string "$a_{k-1}a_{k-2}\ldots a_1a_0$", where $0 \le a_i < b$, for each $i \in [0, k-1]$ denotes the integer $\sum_{i=0}^{k-1} a_i b^i$.

Problem 5.7: Write a function that performs base conversion. Specifically, the input is an integer base b_1, a string s, representing an integer x in base b_1, and another integer base b_2; the output is the string representing the integer x in base b_2. Assume $2 \le b_1, b_2 \le 16$. Use "A" to represent 10, "B" for 11, ..., and "F" for 15. *pg. 177*

5.8 Spreadsheet column encoding

Widely deployed spreadsheets use an alphabetical encoding of the successive columns. Specifically, consecutive columns are identified by "A", "B", "C", ..., "X", "Y", "Z", "AA", "AB", ..., "ZZ", "AAA", "AAB", ...

Problem 5.8: Write a function that converts Excel column ids to the corresponding integer, with "A" corresponding to 1. The function signature is `int ssDecodeColID(string)`; you may ignore error conditions, such as `col` containing characters outside of $[A, Z]$. How would you test your code? *pg. 178*

5.9 Elias gamma coding

A numeral system is a way of writing numbers. The simplest numeral system is the unary numeral system, in which every natural number is represented by a corresponding number of symbols. If the symbol | is chosen, for example, then the number seven would be represented by |||||||.

The Elias gamma code is used to encode positive integers. It is useful when an *a priori* upper bound on the integers being encoded is not known.

Specifically, the Elias gamma code of a positive integer n is computed as follows.

- Write n in binary to form string b.
- Subtract 1 from the number of bits written in the first step, and add that many zeroes to the beginning of string b.

For example, the binary representation of 13 is 1101, which takes four bits to write. Hence the Elias gamma code for 13 is 0001101.

Problem 5.9: Let A be an array of n integers. Write an encode function that returns a string representing the concatenation of the Elias gamma codes for $\langle A[0], A[1], \ldots, A[n-1] \rangle$ in that order, and a decode function that takes a string s assumed to be generated by the encode function, and returns the array that was passed to the encode function. *pg. 178*

5.10 Greatest common divisor

The greatest common divisor (GCD) of positive integers x and y is the largest integer d such that $d \mid x$ and $d \mid y$, where $a \mid b$ denotes a divides b, i.e., $b \bmod a = 0$.

Problem 5.10: Design an algorithm for computing the GCD of two numbers without using multiplication, division or the modulus operators. *pg. 179*

5.11 Enumerating primes

A natural number is called a prime if it is bigger than 1 and has no divisors other than 1 and itself.

Problem 5.11: Write a function that takes a single positive integer argument n ($n \geq 2$) and return all the primes between 1 and n. *pg. 180*

5.12 Checking if rectangles intersect

Call a rectangle R whose sides are parallel to the x-axis and y-axis *xy-aligned*. Such a rectangle is characterized by its left-most lower point (R_x, R_y), its width R_w and its height R_h.

Problem 5.12: Let R and S be xy-aligned rectangles in the Cartesian plane. Write a function which tests if R and S have a nonempty intersection. If the intersection is nonempty, return the rectangle formed by their intersection. *pg. 181*

5.13 Computing $x \times y$ without multiply or add

Often the processors used in embedded systems do not have a hardware multiplier. A program that needs to perform multiplication must do so explicitly.

Problem 5.13: Write a function that multiplies two unsigned positive integers. The only operators you are allowed to use are assignment and the bitwise operators, i.e., », «, |, &, ~, ^. (In particular, you cannot use increment or decrement.) You may use loops, conditionals and functions that you write yourself; other functions are allowed. *pg. 182*

5.14 Computing x/y (⚙)

Problem 5.14: Given two positive integers x and y, how would you compute x/y if the only operators you can use are addition, subtraction, and multiplication? *pg. 182*

CHAPTER

6

Arrays and Strings

The machine can alter the scanned symbol and its behavior is in part determined by that symbol, but the symbols on the tape elsewhere do not affect the behavior of the machine.

— *"Intelligent Machinery,"*
A. M. TURING, 1948

Arrays

The simplest data structure is the *array*, which is a contiguous block of memory. Given an array A which holds n objects, $A[i]$ denotes the i-th object stored in the array. Retrieving and updating $A[i]$ takes $O(1)$ time. However the size of the array is fixed, which makes adding more than n objects impossible. Deletion of the object at location i can be handled by having an auxiliary Boolean associated with the location i indicating whether the entry is valid.

Insertion of an object into a full array can be handled by allocating a new array with additional memory and copying over the entries from the original array. This makes the worst-case time of insertion high but if the new array has, for example, twice the space of the original array, the average time for insertion is constant since the expense of copying the array is infrequent. This concept is formalized using amortized analysis.

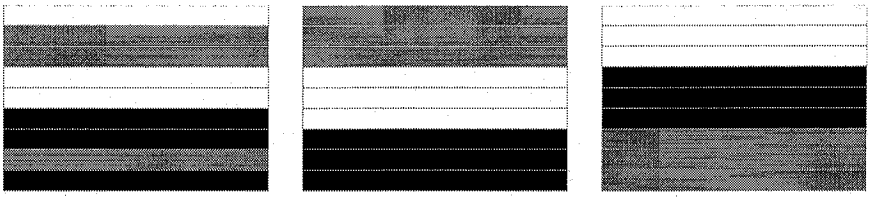
6.1 DUTCH NATIONAL FLAG

The quicksort algorithm for sorting arrays proceeds recursively—it selects an element x (the "pivot"), reorders the array to make all the elements less than or equal to x appear first, followed by all the elements greater than x. The two subarrays are then sorted recursively.

Implemented naïvely, this approach leads to large run times on arrays with many duplicates. One solution is to reorder the array so that all elements less than x appear first, followed by elements equal to x, followed by elements greater than x. This is known as Dutch national flag partitioning, because the Dutch national flag consists of three horizontal bands, each in a different color. Assuming that black precedes white and white precedes gray, Figure 6.1(b) on the facing page is a valid partitioning for Figure 6.1(a) on the next page. If gray precedes black and black precedes white,

Figure 6.1(c) on the facing page is a valid partitioning for Figure 6.1(a) on the next page.

When an array consists of entries from a small set of keys, e.g., $\{0, 1, 2\}$, one way to sort it is to count the number of occurrences of each key. Consequently, enumerate the keys in sorted order and write the corresponding number of keys to the array. If a BST is used for counting, the time complexity of this approach is $O(n \log k)$, where n is the array length and k is the number of keys. This is known as counting sort. Counting sort, as just described, does not differentiate among different objects with the same key value. This problem is concerned with a special case of counting sort when entries are objects rather than keys. Problem 13.4 on Page 99 addresses the general problem.



(a) Before partitioning. (b) A three-way partitioning resembling the Dutch national flag. (c) Another three-way partitioning: the Russian national flag.

Figure 6.1: Illustrating the Dutch national flag problem.

Problem 6.1: Write a function that takes an array A and an index i into A, and rearranges the elements such that all elements less than $A[i]$ appear first, followed by elements equal to $A[i]$, followed by elements greater than $A[i]$. Your algorithm should have $O(1)$ space complexity and $O(|A|)$ time complexity. *pg. 183*

6.2 Lazy initialization (☠)

You have some code which allocates a Boolean-valued array A. The memory manager allocates this array in $O(1)$ time, but the contents of the allocated array are arbitrary. You would like to initialize all the entries to 0. The length n of A is potentially huge and you want to avoid the $O(n)$ time complexity of initialization.

One way to check if an array entry has been written to is to store the indices that have been written to in a hash table. Suppose the drawbacks of a hash table—poor performance if the hash codes are not spread out, and the need for rehashing—are not acceptable.

Problem 6.2: Design a deterministic scheme by which reads and writes to an uninitialized array can be made in $O(1)$ time. You may use $O(n)$ additional storage; reads to uninitialized entry should return `false`. *pg. 184*

6.3 Max difference

The problem of computing the maximum difference in an array, specifically $\max_{i>j}(A[i] - A[j])$ arises in a number of contexts. We introduced this problem

in the context of historical stock quote information on Page 1. Here we study another application of the same problem.

A robot needs to travel along a path that includes several ascents and descents. When it goes up, it uses its battery to power the motor and when it descends, it recovers the energy which is stored in the battery. The battery recharging process is ideal: on descending, every Joule of gravitational potential energy converts to a Joule of electrical energy which is stored in the battery. The battery has a limited capacity and once it reaches this capacity, the energy generated in descending is lost.

Problem 6.3: Design an algorithm that takes a sequence of n three-dimensional coordinates to be traversed, and returns the minimum battery capacity needed to complete the journey. The robot begins with a fully charged battery. *pg. 185*

6.4 Generalizations of max difference

Problem 6.3 on the preceding page, which is concerned with computing $\max_{0 \le i < j \le n-1}(A[j] - A[i])$, generalizes naturally to the following three problems.

Problem 6.4: For each of the following, A is an integer array of length n.

(1.) Compute the maximum value of $(A[j_0] - A[i_0]) + (A[j_1] - A[i_1])$, subject to $i_0 < j_0 < i_1 < j_1$.

(2.) Compute the maximum value of $\sum_{t=0}^{k-1}(A[j_t] - A[i_t])$, subject to $i_0 < j_0 < i_1 < j_1 < \cdots < i_{k-1} < j_{k-1}$. Here k is a fixed input parameter.

(3.) Repeat Problem (2.) when k can be chosen to be any value from 0 to $\lfloor n/2 \rfloor$.

pg. 186

6.5 Subset summing to 0 mod n

Let A be an array of n integers, not necessarily distinct. Let $I = \{i_0, i_1, \ldots, i_{k-1}\}$ be a subset of the indices of A where $k \le n$. Define the subset sum for I to be $\Sigma_{j \in I} A[j]$.

In the 0 mod n-sum subset problem, the input is a nonempty array A. The problem calls for finding a nonempty subset of the indices of A whose subset sum is 0 modulo n. For example, for the array in Figure 6.2, $A[3] + A[4] + A[9] \bmod 10 = 0$, and $\{3, 4, 9\}$ is a corresponding subset. (There are other such subsets.)

Although the problem of finding a subset whose sum is 0 mod k for general k is known to be NP-complete, the 0 mod n-sum subset problem can be solved efficiently. (Note that n is the length of the underlying array as well as the divisor.) In particular there always exists a subset with the desired property.

429	334	62	711	704	763	98	733	721	995
$A[0]$	$A[1]$	$A[2]$	$A[3]$	$A[4]$	$A[5]$	$A[6]$	$A[7]$	$A[8]$	$A[9]$

Figure 6.2: An instance of the 0 mod n-sum subset problem.

Problem 6.5: Design an efficient algorithm for the 0 mod n-sum subset problem. *pg. 187*

6.6 Longest contiguous increasing subarray (☹)

An array is increasing if each element is less than its succeeding element except for the last element.

Problem 6.6: Design and implement an algorithm that takes as input an array A of n elements, and returns the beginning and ending indices of a longest increasing subarray of A. *pg. 187*

6.7 Computing equivalence classes (☹)

Formally, an *equivalence relation* E on a set S is a subset of $S \times S$ that is reflexive (for all $x, (x,x) \in E$), symmetric (for all x and for all $y, (x,y) \in E$ iff $(y,x) \in E$), and transitive (for all x, for all y, and for all z, $(x,y) \in E$ and $(y,z) \in E$ implies $(x,z) \in E$). A *partition* on a set S is a collection of subsets $P = \{S_0, S_1, \ldots, S_{k-1}\}$ having the property that $\cup_{i=0}^{k-1} S_i = S$ and $S_i \cap S_j = \emptyset$ if $i \neq j$. Each subset is referred to as an *equivalence class*. An equivalence relation E on S naturally implies a partition: the equivalence classes are maximal subsets of elements all of which are equivalent under E to one another.

Let $S = \{0, 1, \ldots, n-1\}$. Let A and B be two arrays of length m whose entries are integers from S. These arrays are used to specify equivalence information; in particular, that $A[k]$ and $B[k]$ are equivalent. The *weakest implied equivalence relation* is the equivalence relation in which $x, y \in S$ are assigned to the same equivalence class iff the given equivalence information forces them to be equivalent. For example, if $n = 7$, $A = [1,5,3,6]$, and $B = [2,1,0,5]$, then the weakest implied equivalence relation is $\{\{0,3\}, \{1,2,5,6\}, \{4\}\}$.

Problem 6.7: How would you compute the weakest implied equivalence relation given n, A, and B? You do not have access to any data structure libraries. *pg. 188*

6.8 Offline minimum (☹)

Let σ be a sequence of length n whose elements are drawn from $\mathcal{Z}_n = \{0, 1, 2, \ldots, n-1\}$. No element is repeated, which implies that each integer in $\mathcal{Z}_n$ appears exactly once, i.e., σ is a permutation. We read the elements of σ one at a time, storing them in a set S, starting with the first element. We extract the minimum element from S after $i_0 \leq i_1 \leq \cdots \leq i_{m-1}$-th elements have been read.

Problem 6.8: Suppose you know the permutation σ and the extract sequence $\langle i_0, i_1, \ldots, i_{m-1} \rangle$ in advance. How would you efficiently compute the order in which the m elements are removed from S? *pg. 189*

6.9 BigInteger multiplication

Certain applications require arbitrary precision arithmetic. One way to achieve this is to use strings to represent integers, e.g., with one digit or negative sign per character entry, with the most significant digit appearing first.

Problem 6.9: Write a function that takes two strings representing integers, and returns an integer representing their product. *pg. 190*

6.10 Permuting the elements of an array

A permutation of length n is a one-to-one onto mapping σ from $\{0, 1, \ldots, n-1\}$ to itself. We can represent a permutation using an array Π: set $\Pi[i] = \sigma(i)$. A permutation can be applied to an array A of n elements: $\Pi(A)$ is defined by $\Pi(A[i]) = A[\Pi[i]]$ for $0 \leq i \leq n-1$. It is simple to apply a permutation to a given array if additional storage is available to write the resulting array.

Problem 6.10: Given an array A of n elements and a permutation Π, compute $\Pi(A)$ using only constant additional storage. *pg. 192*

6.11 Invert a permutation

Every one-to-one onto mapping is invertible, i.e., if f is one-to-one onto, then there exists a unique function f^{-1} such that $f^{-1}(f(x)) = x$. In particular, for any permutation Π, there exists a unique permutation Π^{-1} that is the inverse of Π.

Given a permutation represented by an array A, you can compute its inverse B by simply assigning $B[A[i]] = i$ for all values of i.

Problem 6.11: Given an array A of integers representing a permutation Π, update A to represent Π^{-1} using only constant additional storage. *pg. 193*

6.12 Next permutation

A permutation of a set of n elements can be represented using a vector of n integers from $\{0, 1, \ldots, n-1\}$, each one appearing once. There exist exactly $n!$ permutations of n elements. These can be totally ordered using the *lexicographic ordering*—$p \prec_{\text{lex}} q$ if in the first place where p and q differ in their vector representations, the corresponding entry for p is less than that for q. For example, $(2, 0, 1) \prec_{\text{lex}} (2, 1, 0)$.

Problem 6.12: Given a permutation p represented as a vector, return the vector corresponding to the next permutation under lexicographic ordering. If p is the last permutation, return empty vector. For example, if $p = (1, 0, 3, 2)$, your function should return $(1, 2, 0, 3)$. *pg. 193*

6.13 Rotate an array

Let A be an array of n elements. If we have enough memory to make a copy of A, rotating A by i positions is trivial; we just compute $B[j] = A[(i + j) \bmod n]$. If we are given a constant amount of additional memory c, we can rotate the string by c positions a total of $k = \lceil n/c \rceil$ times but this increases the time complexity to $\Theta(nk)$. You cannot use the `rotate` library function in C++. (Imagine you are implementing the library function `rotate`.)

Problem 6.13: Design a $\Theta(n)$ algorithm for rotating an array A of n elements to the right by i positions. You are allowed $O(1)$ additional storage. *pg. 194*

6.14 Sudoku checker

Sudoku is a popular logic-based combinatorial number placement puzzle. The objective is to fill a 9×9 grid with digits subject to the constraint that each column, each row, and each of the nine 3×3 sub-grids that compose the grid contains unique integers in $[1, 9]$. The grid is initialized with a partial assignment as shown in Figure 6.3(a); a partial solution is shown in Figure 6.3(b).

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

(a) Partial assignment.

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

(b) A complete solution.

Figure 6.3: Sudoku configurations.

Problem 6.14: Check whether a 9×9 2D array representing a partially completed Sudoku is valid. Specifically, check that no row, column, and 3×3 2D subarray contains duplicates. A 0-value in the 2D array indicates that entry is blank; every other entry is in $[1, 9]$. *pg. 197*

6.15 Print 2D array in spiral order

An $n \times n$ 2D array A of integers can be written as a sequence of integers in several orders—the most natural ones being row-by-row or column-by-column. In this problem we explore the problem of writing the 2D array in spiral order. For example, the answer of the 2D array in Figure 6.4 should be "1 2 3 6 9 8 7 4 5".

1	2	3
4	5	6
7	8	9

Figure 6.4: A spiral 2D array example.

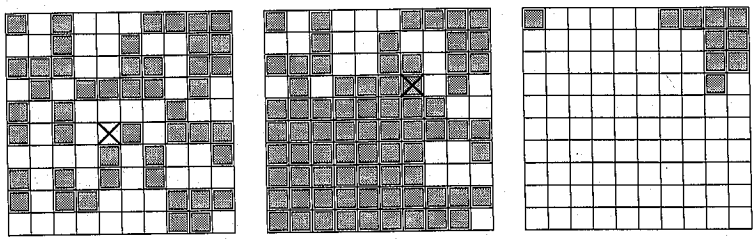
Problem 6.15: Implement a function which takes a 2D array A and prints A in spiral order. *pg. 198*

6.16 Painting

Let A be a $D \times D$ Boolean 2D array encoding a black-and-white image. The entry $A(a, b)$ can be viewed as encoding the color at location (a, b). Define a path from entry (a, b) to entry (c, d) to be a sequence of entries $\langle (x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n) \rangle$ such that

- $(a, b) = (x_1, y_1), (c, d) = (x_n, y_n)$, and
- for each i, $1 \leq i < n$, we have $|x_i - x_{i+1}| + |y_i - y_{i+1}| = 1$.

Define the region associated with a point (i, j) to be all points (i', j') such that there exists a path from (i, j) to (i', j') in which all entries are the same color. In particular this implies (i, j) and (i', j') must be the same color.



(a) (b) (c)

Figure 6.5: The color of all squares associated with the first square marked with a × in (a) have been recolored to yield the coloring in (b). The same process yields the coloring in (c).

Problem 6.16: Implement a routine that takes a $D \times D$ Boolean array A together with an entry (x, y) and flips the color of the region associated with (x, y). See Figure 6.5 for an example of flipping. *pg. 199*

6.17 2D array rotation

Image rotation is a fundamental operation in computer graphics. Figure 6.6 illustrates the rotation operation on a 2D array representing a bit-map of an image. Specifically, the image is rotated by 90 degrees clockwise.

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

(a) Initial 4 × 4 2D array.

13	9	5	1
14	10	6	2
15	11	7	3
16	12	8	4

(b) Array rotated by 90 degrees clockwise.

Figure 6.6: Example of 2D array rotation.

Problem 6.17: Design an algorithm that rotates a $n \times n$ 2D array by 90 degrees clockwise. Assume that $n = 2^k$ for some positive integer k. What is the time complexity of your algorithm? *pg. 200*

Strings

Strings are ubiquitous in programming today—scripting, web development, and bioinformatics all make extensive use of strings. You should know how strings are represented in memory, and understand basic operations on strings such as comparison, copying, joining, splitting, matching, etc. Problems 6.18 to 6.23 on Pages 59–60 are representative of string-related questions. We now present problems on strings which can be solved using elementary techniques. Advanced string processing algorithms often use hash tables (Chapter 12) and dynamic programming (Page 117).

6.18 Run-length encoding

Run-length encoding (RLE) compression offers a fast way to do efficient on-the-flight compression and decompression of strings. The idea is simple—encode successive repeated characters by the repetition count and the character. For example, the RLE of "aaaabcccaa" is "4a1b3c2a". The decoding of "3e4f2e" returns "eeeffffee".

Problem 6.18: Implement run-length encoding and decoding functions. Assume the string to be encoded consists of letters of the alphabet, with no digits, and the string to be decoded is a valid encoding. *pg. 201*

6.19 Reverse all the words in a sentence

Given a string containing a set of words separated by white space, we would like to transform it to a string in which the words appear in the reverse order. For example, "Alice likes Bob" transforms to "Bob likes Alice". We do not need to keep the original string.

Problem 6.19: Implement a function for reversing the words in a string. Your function should use $O(1)$ space. *pg. 202*

6.20 Find the first occurrence of a substring

A good string search algorithm is fundamental to the performance of many applications. Several clever algorithms have been proposed for string search, each with its own trade-offs. As a result, there is no single perfect answer. If someone asks you this question in an interview, the best way to approach this problem would be to work through one good algorithm in detail and discuss at a high level other algorithms.

Problem 6.20: Given two strings s (the "search string") and t (the "text"), find the first occurrence of s in t. *pg. 203*

6.21 Replace and remove

Consider the following two rules that are to be applied to strings over the alphabets {"a", "b", "c", "d"}.

1. Replace each "a" by "dd".
2. Delete each "b".

It is straightforward to implement a function that takes a string s as an argument, and applies these rules to s if the function can allocate $O(|s|)$ additional storage.

Problem 6.21: Write a function which takes as input a string s, and removes each "b" and replaces each "a" by "dd". Use $O(1)$ additional storage—assume s is stored in an array that has enough space for the final result. *pg. 204*

6.22 Phone number mnemonic

Each digit, apart from 0 and 1, in a phone keypad corresponds to one of three or four letters of the alphabet, as shown in Figure 6.7. Since words are easier to remember than numbers, it is natural to ask if a 7 or 10-digit phone number can be represented by a word. For example, "2276696" corresponds to "ACRONYM" as well as "ABPOMZN".

Figure 6.7: Phone keypad.

Problem 6.22: Given a cell phone keypad (specified by a mapping M that takes individual digits and returns the corresponding set of characters) and a number sequence, return all possible character sequences (not just legal words) that correspond to the number sequence. *pg. 205*

6.23 Regular expression matching

A regular expression is a sequence of characters that defines a set of matching strings. For this problem we define a simple subset of a full regular expression language.

A simple regular expression (SRE) is an alphanumeric character, the metacharacter . (dot), an alphanumeric character or dot followed by the metacharacter * (star), or the concatenation of two simple regular expressions. For example, a, aW, aW.9, aW.9*, and aW.*9* are simple regular expressions.

An extended simple regular expression (ESRE) is an SRE, an SRE prepended with the metacharacter ^, an SRE ended with the metacharacter $, or an SRE prepended

with ^ and ended with $. The previous SRE examples are ESREs, as are ^a, aW$, and ^aW.9*$.

First we define what it means for an SRE r to strictly match a string s. Recall s^k denotes the k-th suffix of s.

- If r begins with an alphanumeric character and the next character in r is not star, then r strictly matches s if that same character is at the start of s, and r^1 strictly matches s^1.
- If r begins with an alphanumeric character and the next character in r is star, then r strictly matches s if s can be written as $s1$ concatenated by $s2$, where $s1$ consists of zero or more of the same character, and $s2$ strictly matches r^2.
- If r begins with dot and the next character in r is not star, then r strictly matches s if r^1 strictly matches s^1.
- If r begins with dot and the next character in r is star, then r strictly matches s if s can be written as $s1$ concatenated with $s2$, where $s1$ is of length 0 or more, and r^2 strictly matches $s2$.

Now we define when an ESRE matches a string. Conceptually, the metacharacters ^ and $ stand for the beginning and end of the string, respectively. An ESRE r that does not start with ^ or end with $ matches s if there is a substring t of s such that r strictly matches t.

An ESRE r beginning with ^ matches s if there is a prefix $s1$ of s such that r strictly matches $s1$. An ESRE r ending with $ matches s if there is a suffix $s2$ of s such that r strictly matches $s2$.

The following examples are all concerned with ESREs. aW9 matches any string containing aW9 as a substring. ^aW9 matches aW9 only at the start of a string. aW9$ matches aW9 only at the end of a string. ^aW9$ matches aW9 and nothing else. a.9 matches a89 and xyaW9123 but not aw89. a.*9 matches aw89, and aw*9 matches aww9.

Problem 6.23: Design an algorithm that takes a string s and a string r, assumed to be a well-formed ESRE, and checks if r matches s. *pg. 206*

CHAPTER

7 Linked Lists

The S-expressions are formed according to the following recursive rules.

1. *The atomic symbols p_1, p_2, etc., are S-expressions.*
2. *A null expression Λ is also admitted.*
3. *If e is an S-expression so is (e).*
4. *If e_1 and e_2 are S-expressions so is (e_1, e_2).*

— *"Recursive Functions Of Symbolic Expressions,"*
J. McCarthy, 1959

A *singly linked list* is a data structure that contains a sequence of nodes such that each node contains an object and a reference to the next node in the list. The first node is referred to as the *head* and the last node is referred to as the *tail*; the tail's next field is a reference to `null`. The structure of a singly linked list is given in Figure 7.1. There are many variants of linked lists, e.g., in a *doubly linked list*, each node has a link to its predecessor; similarly, a sentinel node or a self-loop can be used instead of `null`. The structure of a doubly linked list is given in Figure 7.2. Since lists can be defined recursively, recursion is a natural candidate for list manipulation.

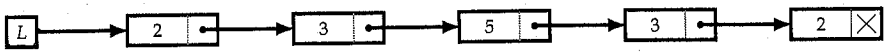


Figure 7.1: Example of a singly linked list.

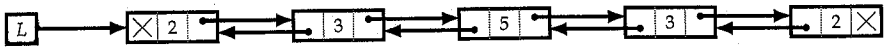


Figure 7.2: Example of a doubly linked list.

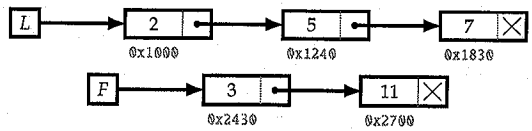
For all problems in this chapter, unless otherwise stated, L is a singly linked list, and your solution may not use more than a few words of storage, regardless of the length of the list. Specifically, each node has two entries—a `data` field, and a `next` field, which points to the next node in the list, with the next field of the last node being `null`. Its prototype in C++ is listed as follows:

```
template <typename T>
class node_t {
  public:
```

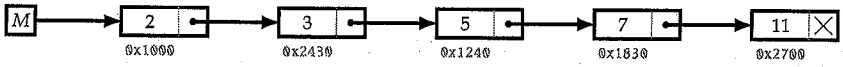
```
    T data;
    shared_ptr<node_t<T>> next;
};
```

7.1 Merge two sorted lists

Let L and F be singly linked lists of numbers. Assume the numbers in L and F appear in sorted order within the lists. The *merge* of L and F is a list consisting of the nodes of L and F in which keys appear in sorted order. The merge function is shown in Figure 7.3.



(a) Two sorted lists.



(b) The merge of the two lists in (a).

Figure 7.3: Merging sorted lists.

Problem 7.1: Write a function that takes L and F, and returns the merge of L and F. Your code should use $O(1)$ additional storage—it should reuse the nodes from the lists provided as input. Your function should use $O(1)$ additional storage, as illustrated in Figure 7.3. The only field you can change in a node is `next`. *pg. 207*

7.2 Checking for cyclicity

Although a linked list is supposed to be a sequence of nodes ending in a `null`, it is possible to create a cycle in a linked list by making the `next` field of an element reference to one of the earlier nodes.

Problem 7.2: Given a reference to the head of a singly linked list L, how would you determine whether L ends in a `null` or reaches a cycle of nodes? Write a function that returns `null` if there does not exist a cycle, and the reference to the start of the cycle if a cycle is present. (You do not know the length of the list in advance.) *pg. 208*

7.3 Median of a sorted circular linked list

It is relatively straightforward to find the median of a sorted linked list in $O(n)$ time. However, this problem becomes trickier if the list is circular.

Problem 7.3: Write a function that takes a sorted circular singly linked list and a pointer to an arbitrary node in this linked list, and returns the median of the linked list. *pg. 210*

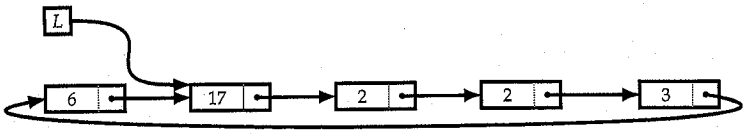


Figure 7.4: Example of a sorted circular linked list.

7.4 Overlapping lists—no lists have cycle

Given two singly linked lists, $L1$ and $L2$, there may be list nodes that are common to both $L1$ and $L2$. (This may not be a bug—it may be desirable from the perspective of reducing memory footprint, as in the flyweight pattern, or maintaining a canonical form.) For example, $L1$ and $L2$ in Figure 7.5 overlap at Node I.

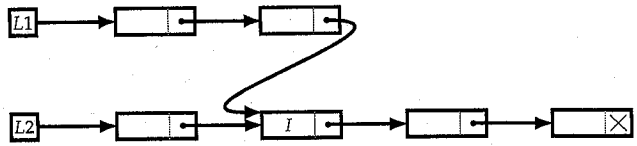


Figure 7.5: Example of overlapping lists.

Problem 7.4: Let $h1$ and $h2$ be the heads of lists $L1$ and $L2$, respectively. Assume that $L1$ and $L2$ are well-formed, that is each consists of a finite sequence of nodes. (In particular, neither list has a cycle.) How would you determine if there exists a node r reachable from both $h1$ and $h2$ by following the next fields? If such a node exists, find the node that appears earliest when traversing the lists. You are constrained to use no more than constant additional storage. *pg. 211*

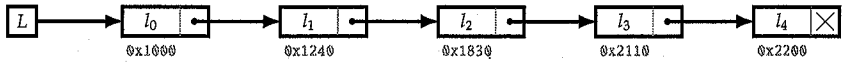
7.5 Overlapping lists—lists may have cycles

Problem 7.5: Solve Problem 7.4 for the case where $L1$ and $L2$ may each or both have a cycle. If such a node exists, return a node that appears first when traversing the lists. This node may not be unique—if $L1$ has a cycle $\langle n_0, n_1, \ldots, n_{k-1}, n_0 \rangle$, where n_0 is the first node encountered when traversing $L1$, then $L2$ may have the same cycle but a different first node. *pg. 212*

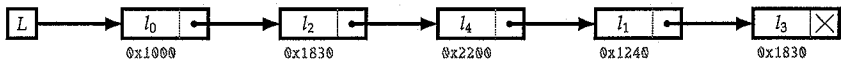
7.6 Even-odd merge

Let $L = \langle l_0, l_1, l_2, \ldots, l_{n-1} \rangle$ be a sequence. Define even-odd(L) to be the sequence $\langle l_0, l_2, l_4, \ldots, l_1, l_3, \ldots \rangle$, i.e., the elements at even indices followed by the elements at odd indices. The even-odd merge function is shown in Figure 7.6 on the facing page.

Problem 7.6: Write a function that takes a singly linked list L, and reorders the elements of L so that the new list represents even-odd(L). Your function should use



(a) Initial list L. The number in hex below a node indicates the memory address of that node.



(b) even-odd(L)—note that no new nodes have been allocated.

Figure 7.6: Even-odd merge example.

$O(1)$ additional storage, as illustrated in Figure 7.6. The only field you can change in a node is `next`. *pg. 213*

7.7 Deletion from a singly linked list

Given a node in a singly linked list, deleting it in $O(1)$ time appears impossible because its predecessor's next field has to be updated. Surprisingly, it can be done with one small caveat—the node to delete cannot be the last one in the list and it is easy to copy the value part of a node.

Problem 7.7: Let v be a node in a singly linked list L. Node v is not the tail; delete it in $O(1)$ time. *pg. 214*

7.8 Remove the k-th last element from a list

Without knowing the length of a linked list, it is not trivial to delete the k-th last element in a singly linked list.

Problem 7.8: Given a singly linked list L and a number k, write a function to remove the k-th last element from L. Your algorithm cannot use more than a few words of storage, regardless of the length of the list. In particular, you cannot assume that it is possible to record the length of the list. *pg. 214*

7.9 Reversing a singly linked list

Suppose you were given a singly linked list L of integers sorted in ascending order and you need to return a list with the elements sorted in descending order. Memory is scarce, but you can reuse nodes in the original list, i.e., your function can change L.

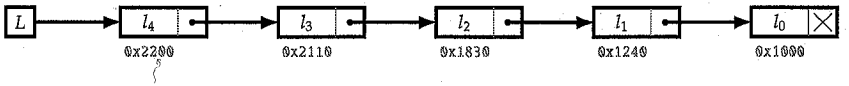


Figure 7.7: The reversed list for the list in Figure 7.6(a). Note that no new nodes have been allocated.

Problem 7.9: Give a linear time non-recursive function that reverses a singly linked list. The function should use no more than constant storage beyond that needed for the list itself. The desired transformation is illustrated in Figure 7.7. *pg. 215*

7.10 Palindromicity in linked list

It is straightforward to check whether the sequence stored in an array is a palindrome. However, if this sequence is stored as a singly linked list, the problem of detecting palindromicity becomes more challenging. See Figure 7.1 on Page 62 for an example of a palindromic singly linked list.

Problem 7.10: Write a function that determines whether a sequence represented by a singly linked list L is a palindrome. Assume L can be changed and does not have to be restored it to its original state. *pg. 216*

7.11 Zipping a list

Let $L = \langle l_0, l_1, l_2, \ldots, l_{n-1} \rangle$. Define zip($L$) to be $\langle l_0, l_{n-1}, l_1, l_{n-2}, \ldots \rangle$. The zip function is shown in Figure 4.1 on Page 25.

Problem 7.11: Write a function that takes a singly linked list L, and reorders the elements of L to form a new list representing zip(L). Your function should use $O(1)$ additional storage, as illustrated in Figure 4.1 on Page 25. The only field you can change in a node is `next`. *pg. 216*

7.12 Copying a postings list

In a "postings list" each node has a data field, a field for the next pointer, and a jump field—the jump field points to any other node. The last node in the postings list has next set to `null`; all other nodes have non-`null` next and jump fields. For example, Figure 7.8 is a postings list with four nodes.

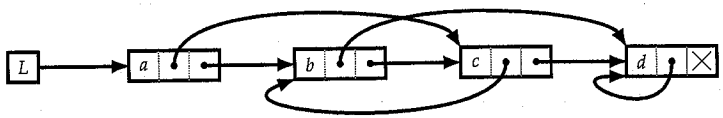


Figure 7.8: A postings list.

Problem 7.12: Implement a function which takes as input a pointer to the head of a postings list L, and returns a copy of the postings list. Your function should take $O(n)$ time, where n is the length of the postings list and should use $O(1)$ storage beyond that required for the n nodes in the copy. You can modify the original list, but must restore it to its initial state before returning. *pg. 217*

CHAPTER

8

Stacks and Queues

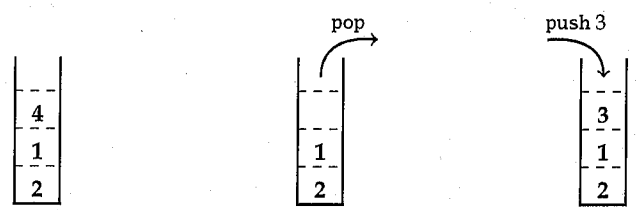
Linear lists in which insertions, deletions, and accesses to values occur almost always at the first or the last node are very frequently encountered, and we give them special names . . .

— *"The Art of Computer Programming, Volume 1,"*
D. E. KNUTH, 1997

Stacks

The *stack* ADT supports two basic operations—push and pop. Elements are added (pushed) and removed (popped) in last-in, first-out order, as shown in Figure 8.1. If the stack is empty, pop typically returns a `null` or throws an exception.

When the stack is implemented using a linked list these operations have $O(1)$ time complexity. If it is implemented using an array, there is maximum number of entries it can have—push and pop are still $O(1)$. If the array is dynamically resized, the amortized time for both push and pop is $O(1)$. A stack can support additional operations such as peek (return the top of the stack without popping it).



(a) Initial configuration. (b) Perform pop on (a). (c) Perform push 3 on (b).

Figure 8.1: Operations on a stack.

8.1 STACK WITH MAX OPERATION

Problem 8.1: Design a stack that supports a `max` operation, which returns the maximum value stored in the stack, and throws an exception if the stack is empty. Assume elements are comparable. All operations must be $O(1)$ time. You can use $O(n)$ additional space, beyond what is required for the elements themselves. *pg. 219*

8.2 Expression evaluation

A string is said to be an arithmetical expression in Reverse Polish notation (RPN) if:

(1.) It is a single digit or a sequence of digits.

(2.) It is of the form "$A, B, \circ$" where A and B are RPN expressions and $\circ$ is one of $+, -, \times, /$.

(3.) It is of the form "$-A$" where A is an RPN expression.

For example, the following strings satisfy these rules: "$3, 4, \times, 1, 2, +, +$", "$1, 1, +, -2, \times$", "$4, 6, /, 2, /$".

An RPN expression can be evaluated uniquely to an integer, which is determined recursively. The base case corresponds to Rule (1.), which is an integer expressed in base-10 positional system. Rules (2.) and (3.) on the current page correspond to the recursive cases, and the RPNs are evaluated in the natural way, e.g., if A evaluates to 2 and B evaluates to 3, then "$A, B, \times$" evaluates to 6.

Problem 8.2: Write a function that takes an arithmetical expression in RPN and returns the number that the expression evaluates to. *pg. 221*

8.3 Printing the keys in a BST

BSTs are the subject of Chapter 14. In summary, a BST is a set of nodes. Each node n has a reference to a left child (denoted by *n.left*) and a right child (*n.right*), and a key (*n.key*). Either or both children may be `null`. The node n is referred to as the parent of *n.left* and *n.right*. The keys are from a totally ordered set, and nodes satisfy the BST property—if *n.left* is not `null`, $n.left.key \leq n.key$ and if *n.right* is not `null`, $n.right.key \geq n.key$. Node m is said to be a descendant of n if $m = n.left$ or $m = n.right$, or if m is a descendant of *n.left* or of *n.right*.

Problem 8.3: Given a BST node n, print all the keys at n and its descendants. The nodes should be printed in sorted order, and you cannot use recursion. For example, for Node I in the binary search tree in Figure 14.1 on Page 105 you should print the sequence $\langle 23, 29, 31, 37, 41, 43, 47, 53 \rangle$. *pg. 223*

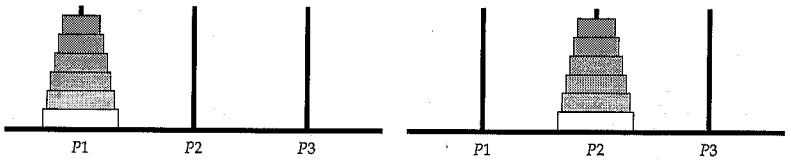
8.4 Searching a postings list

Postings lists are described in Problem 7.12 on Page 66. One way to enumerate the nodes in a postings list is to iteratively follow the next field. Another is to always first follow the jump field if it leads to a node that has not been explored previously, and then search from the next node. Call the order in which these nodes are visited the jump-first order.

Problem 8.4: Write recursive and iterative routines that take a postings list, and computes the jump-first order. Assume each node has an `order` field, which is an integer that is initialized to -1 for each node. *pg. 223*

8.5 Towers of Hanoi

You are given n rings. The i-th ring has diameter i. The rings are initially in sorted order on a peg ($P1$), with the largest ring at the bottom. You are to transfer these rings to another peg ($P2$), which is initially empty. This is illustrated in Figure 8.2. You have a third peg ($P3$), which is initially empty. The only operation you can do is taking a single ring from the top of one peg and placing it on the top of another peg; you must never place a bigger ring above a smaller ring.



(a) Initial configuration. (b) Desired configuration.

Figure 8.2: Towers of Hanoi for $n = 6$.

Problem 8.5: Exactly n rings on $P1$ need to be transferred to $P2$, possibly using $P3$ as an intermediate, subject to the stacking constraint. Write a function that prints a sequence of operations that transfers all the rings from $P1$ to $P2$. *pg. 224*

8.6 Views of the sunset

You are given with a series of buildings that have windows facing west. The buildings are in a straight line, and if a building b is to the east of a building whose height is greater than or equal to b, it is not possible to view the sunset from b.

Problem 8.6: Design an algorithm that processes buildings as they are presented to it and tracks the buildings that have a view of the sunset. The number of buildings is not known in advance. Buildings are given in east-to-west order and are specified by their heights. The amount of memory your algorithm uses should depend solely on the number of buildings that have a view; in particular it should not depend on the number of buildings processed. *pg. 226*

8.7 Stack sorting

Problem 8.7: Design an algorithm to sort a stack S of numbers in descending order. The only operations allowed are `push`, `pop`, `top` (which returns the top of the stack without a `pop`), and `empty`. You cannot explicitly allocate memory outside of a few words. *pg. 226*

8.8 Normalized path names

A file or directory can be specified via a string called the path name. This string may specify an absolute path, starting from the root, e.g., `/usr/bin/gcc`, or a path relative to the current working directory, e.g., `scripts/awkscripts`.

The same directory may be specified by multiple directory paths. For example, `/usr/lib/../bin/gcc` and `scripts//./../scripts/awkscripts/././` specify equivalent absolute and relative path names.

Problem 8.8: Write a function which takes a path name, and returns the shortest equivalent path name. Assume individual directories and files have names that use only alphanumeric characters. Subdirectory names may be combined using forward slashes (/), the current directory (.), and parent directory (..).

The formal grammar is specified as follows:

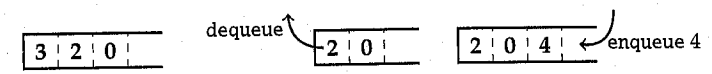
$$\begin{aligned} \texttt{name} &= [A-Za-z0-9]+ \\ \texttt{spdir} &= \texttt{.} \mid \texttt{..} \\ \texttt{pathname} &= \texttt{name} \mid \texttt{spdir} \mid [\texttt{spdir} \mid \texttt{name} \mid \texttt{pathname}]?/+\texttt{pathname}? \end{aligned}$$

Here + denotes one or more repetitions or the preceding token, and ? denotes 0 or 1 occurrences of the preceding token. You should throw an exception on invalid path names. *pg.* 227

Queues

The *queue* ADT supports two basic operations—`enqueue` and `dequeue`. (If the queue is empty, `dequeue` typically returns a `null` or throws an exception.) Elements are added (enqueued) and removed (dequeued) in first-in, first-out order.

A queue can be implemented using a linked list, in which case these operations have $O(1)$ time complexity. Other operations can be added, such as `head` (which returns the item at the start of the queue without removing it), and `tail` (which returns the item at the end of the queue without removing it). A queue can also be implemented using an array; see Problem 8.10 on the next page for details.



(a) Initial configuration. (b) Queue (a) after dequeue. (c) Queue (b) after enqueue(4).

Figure 8.3: Examples of enqueue and dequeue.

A *deque*, also sometimes called a double-ended queue, is a doubly linked list in which all insertions and deletions are from one of the two ends of the list, i.e., at the head or the tail. An insertion to the front is called a `push`, and an insertion to the back is called an `inject`. A deletion from the front is called a `pop`, and a deletion from the back is called an `eject`.

8.9 Printing a binary tree in level order

Binary trees are the subject of Chapter 9. In summary, a binary tree is a root node, which is either `null`, or an object with three fields: a key, a left child, and a right

child. The left and right children are themselves binary trees and are required to be disjoint.

Node d is a descendant of node a iff $d = a$ or d is a child of a or d is a descendant of a child of a. Assign levels to nodes in a binary tree as follows: level(root) = 0, and for any node $c \neq$ root, level(c) = 1 + level(n), where n is the parent of c.

Problem 8.9: Given the root node r of a binary tree, print all the keys and levels at r and its descendants. The nodes should be printed in order of their level. You cannot use recursion. You may use a single queue, and constant additional storage. For example, you should print the sequence $\langle 314, 6, 6, 271, 561, 2, 271, 28, 0, 3, 1, 28, 17, 401, 257, 641 \rangle$ for the binary tree in Figure 9.1 on Page 73. *pg. 228*

8.10 Implement a circular queue

A queue can be implemented using an array and two additional fields, the beginning and the end indices. This structure is sometimes referred to as a circular queue. Both `enqueue` and `dequeue` have $O(1)$ time complexity. If the array is fixed, there is a maximum number of entries that can be stored. If the array is dynamically resized, the total time for m combined `enqueue` and `dequeue` operations is $O(m)$.

Problem 8.10: Implement a queue API using an array for storing elements. Your API should include a constructor function, which takes as argument the capacity of the queue, `enqueue` and `dequeue` functions, a `size` function, which returns the number of elements stored, and implement dynamic resizing. *pg. 229*

8.11 Implement a queue using two unsigned integers

Problem 8.11: Implement a queue using two unsigned integer-valued variables. Assume that the only elements pushed into the queue are integers in $[0, 9]$. Your program should work correctly when 0s are the only elements in the queue. What is the maximum number of elements that can be stored in the queue for it to operate correctly? *pg. 230*

8.12 Queue from two stacks

Queue insertion and deletion follows first-in, first-out semantics; stack insertion and deletion is last-in, first-out. It can be shown rigorously that it is impossible to implement a queue with capacity n (i.e., a queue capable of holding up to n elements at a time) using a stack with capacity n and $O(1)$ additional storage. (The proof, given in Li, *et al.*, *"The Power of the Queue"*, is nontrivial.)

Problem 8.12: How would you implement a queue given two stacks and $O(1)$ additional storage? Your implementation should be efficient—the time to do a sequence of m combined enqueues and dequeues should be $O(m)$. *pg. 231*

8.13 Queue with max

The queue ADT is usually expressed in terms of enqueue and dequeue operations. Suppose the keys are from a totally ordered set, e.g., integers, and we want to support a max operation, which returns the maximum element stored in the queue.

Problem 8.13: How would you implement a queue so that any series of m combined enqueue, dequeue, and max operations can be done in $O(m)$ time? *pg.* 232

8.14 Maximum of a sliding window

Network traffic control sometimes require the maximum traffic volume $m(t, w)$ in the time interval $[t - w, t]$ for each time t in the day, where w is the *window size*. This problem explores the development of an efficient algorithm for computing these maximum traffic volumes.

Problem 8.14: Let A be an array of length n, and w the window size. Entry $A[i]$ is a pair (t_i, v_i), where t_i is the timestamp and v_i the traffic volume at that time. Assume A is sorted by increasing timestamp. Design an algorithm to compute $v_i = \max\{v_j \mid (t_i - t_j) \leq w, j \leq i\}$, for $0 \leq i \leq n - 1$. *pg.* 234

CHAPTER

9 Binary Trees

The method of solution involves the development of a theory of finite automata operating on infinite trees.

— *"Decidability of Second Order Theories and Automata on Trees,"*
M. O. RABIN, 1969

A *binary tree* is a data structure that is useful for representing hierarchy. Formally, a binary tree is a finite set of nodes T that is either empty, or consists of a *root* node r together with two disjoint subsets L and R themselves binary trees whose union with $\{r\}$ equals T. The set L is called the *left binary tree* and R is the *right binary tree* of T. The left binary tree is referred to as the *left child* or the *left subtree* of the root, and the right binary tree is referred to as the *right child* or the *right subtree* of the root.

Figure 9.1 gives a graphical representation of a binary tree. Node A is the root. Nodes B and I are the left and right children of A.

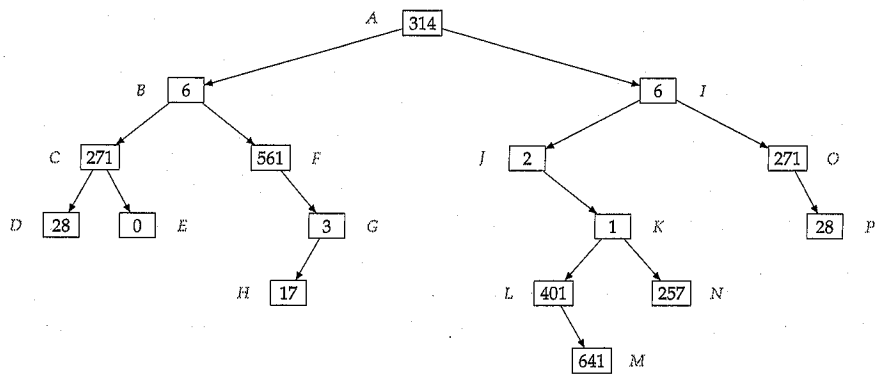


Figure 9.1: Example of a binary tree.

Often the root stores additional data. Its prototype in C++ is listed as follows:

```
template <typename T>
class BinaryTree {
  public:
    T data;
    shared_ptr<BinaryTree<T>> left, right;
};
```

Each node, except the root, is itself the root of a left subtree or a right subtree. If l is the root of p's left subtree, we will say l is the *left child* of p, and p is the *parent* of l; the notion of *right child* is similar. If n is a left or a right child of p, we say it is a *child* of p. Note that with the exception of the root, every node has a unique parent. Often, but not always, the node has a parent field (which is `null` for the root). Observe that for any node n there exists a unique sequence of nodes from the root to n with each subsequent node being a child of the previous node. This sequence is sometimes referred to as the *search path* from the root to n.

The parent-child relationship defines an ancestor-descendant relationship on nodes in a binary tree. Specifically, a is an *ancestor* of d if a lies on the search path from the root to d. If a is an ancestor of d, we say d is a *descendant* of a. Our convention is that x is an ancestor and descendant of itself. A node that has no descendants except for itself is called a *leaf*.

The *depth* of a node n is the number of nodes on the search path from the root to n, not including n itself. The *height* of a binary tree is the maximum depth of any node in that tree.

As concrete examples of these concepts, consider the binary tree in Figure 9.1 on the preceding page. Node I is the parent of J and O. Node G is a descendant of B. The search path to L is $\langle A, I, J, K, L \rangle$. The depth of N is 4. Node M is the node of maximum depth, and hence the height of the tree is 5. The height of the subtree rooted at B is 3. The height of the subtree rooted at H is 0. Nodes $D, E, H, M, N,$ and P are the leaves of the tree.

A *full binary tree* is a binary tree in which every node other than the leaves has two children. A *perfect binary tree* is a full binary tree in which all leaves are at the same depth or same level, and in which every parent has two children. A *complete binary tree* is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible. (This terminology is not universal, e.g., some authors use complete binary tree where we write perfect binary tree.) It is straightforward to prove using induction that the number of non-leaf nodes in a full binary tree is one less than the number of leaves. A perfect binary tree of height h contains exactly $2^{h+1} - 1$ nodes, of which 2^h are leaves. A complete binary tree on n nodes has height $\lfloor \lg n \rfloor$.

A key computation on a binary tree is *visiting* all the nodes in the tree. (Visiting is also sometimes called *walking* or *traversing*.) Here are some ways in which this visit can be done.

- Visit the left subtree, the root, then the right subtree (an *inorder* visit).
- Visit the root, the left subtree, then the right subtree (a *preorder* visit).
- Visit the left subtree, the right subtree, and then the root (a *postorder* visit).

Let T be a binary tree on n nodes, with height h. Implemented recursively, these visits have $O(n)$ time complexity and $O(h)$ additional space complexity. (The space complexity is dictated by the maximum depth of the function call stack.) If each node has a parent field, the visits can be done with $O(1)$ additional space complexity.

Remarkably, an inorder visit can be implemented in $O(1)$ additional space even without parent fields. The approach is based on temporarily setting right child fields

for leaf nodes, and later undoing these changes. Code for this algorithm, known as a *Morris traversal*, is given below. It is largely of theoretical interest; one major shortcoming is that it is not thread-safe, since it mutates the tree, albeit temporarily.

```
template <typename T>
void inorder_traversal(shared_ptr<BinaryTree<T>> n) {
  while (n) {
    if (n->left) {
      // Find the predecessor of n
      shared_ptr<BinaryTree<T>> pre = n->left;
      while (pre->right && pre->right != n) {
        pre = pre->right;
      }

      // Build the successor link
      if (pre->right) {  // pre->right == n
        // Revert the successor link if predecessor's successor is n
        pre->right = nullptr;
        cout << n->data << endl;
        n = n->right;
      } else {  // if predecessor's successor is not n
        pre->right = n;
        n = n->left;
      }
    } else {
      cout << n->data << endl;
      n = n->right;
    }
  }
}
```

The term tree is overloaded, which can lead to confusion; see Page 131 for an overview of the common variants.

9.1 Balanced binary trees

A binary tree is said to be balanced if for each node in the tree, the difference in the height of its left and right subtrees is at most one.

Problem 9.1: Write a function that takes as input the root of a binary tree and returns `true` or `false` depending on whether the tree is balanced. Use $O(h)$ additional storage, where h is the height of the tree. *pg. 235*

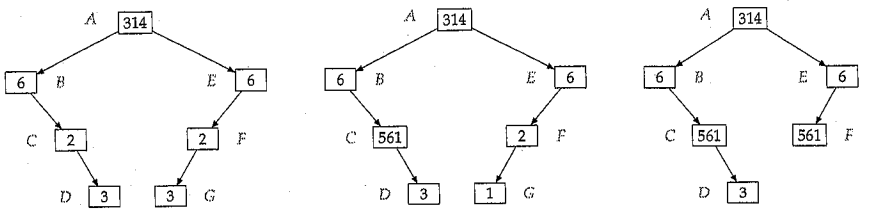
9.2 k-balanced nodes

Define a node in a binary tree to be k-balanced if the difference in the number of nodes in its left and right subtrees is no more than k.

Problem 9.2: Design an algorithm that takes as input a binary tree and positive integer k, and returns a node u in the binary tree such that u is not k-balanced, but all of u's descendants are k-balanced. If no such node exists, return `null`. For example, when applied to the binary tree in Figure 9.1 on Page 73, your algorithm should return Node J if $k = 3$. *pg. 236*

9.3 Symmetric binary tree

A binary tree is symmetric if you can draw a vertical line through the root and then the left subtree is the mirror image of the right subtree. The concept of a symmetric binary tree is illustrated in Figure 9.2.



(a) A symmetric binary tree. (b) An asymmetric binary tree. (c) An asymmetric binary tree.

Figure 9.2: Symmetric and asymmetric binary trees.The tree in (a) is structurally symmetric, but symmetry requires that corresponding nodes have the same keys; here C and F as well as D and G break symmetry. The tree in (c) is asymmetric because there is no node corresponding to G.

Problem 9.3: Write a function that takes as input the root of a binary tree and returns `true` or `false` depending on whether the tree is symmetric. *pg. 237*

9.4 Locking in a binary tree

Problem 9.4: For a certain application, processes need to lock nodes in a binary tree. Implement a library for locking nodes in a binary tree, subject to the constraint that a node cannot be locked if any of its descendants or ancestors are locked. Specifically, write functions `isLock()`, `lock()`, and `unLock()`, with time complexities $O(1)$, $O(h)$, and $O(h)$. Here h is the height of the binary tree. Assume that each node has a parent field. *pg. 238*

9.5 Inorder traversal with $O(1)$ space

The direct implementation of an inorder walk using recursion has $O(h)$ space complexity, where h is the height of the tree. Recursion can be removed with an explicit stack, but the space complexity remains $O(h)$. If the tree is mutable, we can do inorder traversal in $O(1)$ space—this is the Morris traversal described on the preceding page. The Morris traversal does not require that nodes have parent fields.

Problem 9.5: Let T be the root of a binary tree in which nodes have an explicit parent field. Design an iterative algorithm that enumerates the nodes inorder and uses $O(1)$ additional space. Your algorithm cannot modify the tree. *pg. 239*

9.6 Determining the k-th node in an inorder traversal

It is trivial to find the k-th node that appears in an inorder traversal with $O(n)$ time complexity. However, with additional information on each node, you can do better.

Problem 9.6: Design a function that efficiently computes the k-th node appearing in an inorder traversal. Specifically, your function should take as input a binary tree T and an integer k. Each node has a `size` field, which is the number of nodes in the subtree rooted at that node. What is the time complexity of your function? *pg. 240*

9.7 Reconstructing a binary tree from traversal data

Many different binary trees yield the same sequence of keys in an inorder, preorder, or postorder traversal. However, given an inorder traversal and one of any two other traversal orders of a binary tree, there exists a unique binary tree that yields those orders, assuming each node holds a distinct key. For example, the unique binary tree whose inorder traversal sequence is $\langle F, B, A, E, H, C, D, I, G\rangle$ and whose preorder traversal sequence is $\langle H, B, F, E, A, C, D, G, I\rangle$ is given in Figure 9.3.

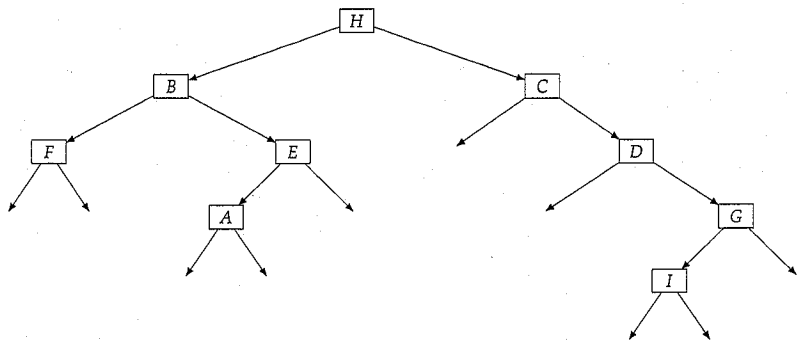


Figure 9.3: A binary tree—edges that do not terminate in nodes denote empty subtrees.

Problem 9.7: Given an inorder traversal order, and one of a preorder or a postorder traversal order of a binary tree, write a function to reconstruct the tree. *pg. 241*

9.8 Reconstructing a binary tree from a preorder traversal with marker

Many possible binary trees on nodes $\{v_0, v_1, \ldots, v_{n-1}\}$ yield the sequence of nodes $\sigma = \langle v_0, v_1, \ldots, v_{n-1}\rangle$ from a preorder walk. Node v_0 must be the root, but the left subtree could consist of $\{v_1, \ldots, v_l\}$ for $l \in [1, n-1]$; it could also be empty.

Suppose, the preorder walk routine is modified to mark where a left or right child was empty. For example, the binary tree in Figure 9.3 is the unique tree that yields the following preorder traversal sequence:

$$\langle H, B, F, \texttt{null}, \texttt{null}, E, A, \texttt{null}, \texttt{null}, \texttt{null}, C, \texttt{null}, D, \texttt{null}, G, I, \texttt{null}, \texttt{null}, \texttt{null}\rangle$$

Problem 9.8: Design an $O(n)$ time algorithm for reconstructing a binary tree from a preorder visit sequence that uses `null` to mark empty children. How would you

modify your reconstruction algorithm if the sequence corresponded to a postorder or inorder walk? *pg.* 242

9.9 Form a linked list from the leaves of a binary tree

In some applications of a binary tree, only the leaf nodes contain actual information. For example, in a single knockout tournament organized as a binary tree, we can link the leaves to get a list of participants.

Problem 9.9: Given a binary tree, write a function which forms a linked list from the leaves of the binary tree. The leaves should appear in left-to-right order. For example, when applied to the binary tree in Figure 9.1 on Page 73, your function should return $\langle D, E, H, M, N, P \rangle$. *pg.* 243

9.10 The exterior of a binary tree (☺)

Problem 9.10: Write a function that prints the nodes on the exterior of a binary tree in anti-clockwise order, i.e., print the nodes on the path from the root to the leftmost leaf in that order, then the leaves from left-to-right, then the nodes from the rightmost leaf up to the root. For example, when applied to the binary tree in Figure 9.1 on Page 73, your function should return $\langle A, B, C, D, E, H, M, N, P, O, I \rangle$. (By leftmost (rightmost) leaf, we mean the leaf that appears first (last) in an inorder walk.) *pg.* 244

9.11 Lowest common ancestor in a binary tree

Any two nodes in a binary tree have a common ancestor, namely the root. The lowest common ancestor (LCA) of any two nodes in a binary tree is the node furthest from the root that is an ancestor of both nodes. For example, the LCA of M and N in Figure 9.1 on Page 73 is K.

Computing the LCA has important applications. For example, in an interval tree (Problem 14.23 on Page 113), the LCA of any two nodes is key to computing the smallest interval that contains the intervals stored at those nodes.

Problem 9.11: Design an efficient algorithm for computing the LCA of nodes a and b in a binary tree in which nodes do not have a parent pointer. *pg.* 245

9.12 Lowest common ancestor in a binary tree, with parent pointer

Problem 9.12: Given two nodes in a binary tree T, design an algorithm that computes their LCA. Assume that each node has a parent pointer. The tree has n nodes and height h. Your algorithm should run in $O(1)$ space and $O(h)$ time. *pg.* 245

9.13 Lowest common ancestor in a binary tree, close ancestor

Problem 9.12 is concerned with computing the LCA in a binary tree with parent pointers in $O(h)$ time and $O(1)$ space. The algorithm presented in Solution 9.12 on

Page 245 entails traversing all the way to the root, so even if $\max(d_a - d_l, d_b - d_l) \ll h$, its time complexity remains $O(h)$.

Problem 9.13: Design an algorithm for computing the LCA of a and b that has time complexity $O(\max(d_a - d_l, d_b - d_l))$. What is the worst-case time and space complexity of your algorithm? *pg. 246*

9.14 Shortest unique prefix (☺)

Formally, a sequence is a function whose domain is of the form $\{0, 1, 2, \ldots, n-1\}$. A sequence f is often written as $\langle f(0), f(1), \ldots, f(n-1)\rangle$. A string is a finite sequence of symbols drawn from an alphabet; the length of the string is the cardinality of its domain. A string may be of length zero—this is referred to as the empty string, and is denoted by ϵ. A prefix of a string s defined on domain $\{0, 1, \ldots, n-1\}$ is either ϵ or the restriction of s to domain $\{0, 1, \ldots, m-1\}$, for $0 \leq m \leq n$.

This problem is concerned with finding the shortest prefix of a string s that is not in a set of strings D. For example:

- If s = "cat" and D = {"dog", "be", "cut"} return "ca".
- If s = "cat" and D = {"dog", "be", "cut", "car"} return "cat".
- If s = "cat" and D = {"dog", "be", "cut", "car", "cat"} return ϵ.

Problem 9.14: Given a string s and a set of strings D, find the shortest prefix of s which is not a prefix of any string in D. *pg. 247*

CHAPTER

10 Heaps

Using F-heaps we are able to obtain improved running times for several network optimization algorithms.

— *"Fibonacci heaps and their uses,"*
M. L. FREDMAN AND R. E. TARJAN, 1987

A *heap* is a specialized binary tree, specifically it is a complete binary tree. It supports $O(\log n)$ insertions, $O(1)$ time lookup for the max element, and $O(\log n)$ deletion of the max element. The extract-max operation is defined to delete and return the maximum element. (The *min-heap* is a completely symmetric version of the data structure and supports $O(1)$ time lookups for the minimum element.)

A max-heap can be implemented as an array; the children of the node at index i are at indices $2i+1$ and $2i+2$. Searching for arbitrary keys has $O(n)$ time complexity. Anything that can be done with a heap can be done with a balanced BST with the same or better time and space complexity but with possibly some implementation overhead. There is no relationship between the heap data structure and the portion of memory in a process by the same name.

10.1 MERGING SORTED FILES

You are given 500 files, each containing stock trade information for an S&P 500 company. A line within a file captures a trade as follows:

```
1232111,AAPL,30,456.12
```

The first number is the time of the trade expressed as the number of milliseconds since the start of the day's trading. Lines within each file are sorted by this value. (The remaining values are the stock symbol, number of shares, and price.) Your task is to create a single file containing all the trades sorted by trade times. The individual files are of the order of 5–100 megabytes; the combined file will be of the order of five gigabytes.

Problem 10.1: Design an algorithm that takes a set of files containing stock trade information in sorted order, and writes a single file containing the lines appearing in the individual files sorted in sorted order. The algorithm should use very little RAM, ideally of the order of a few kilobytes. *pg. 248*

10.2 Sort k-increasing-decreasing array

An array A of n integers is said to be k-increasing-decreasing if elements repeatedly increase up to a certain index after which they decrease, then again increase, a total of k times, as illustrated in Figure 10.1.

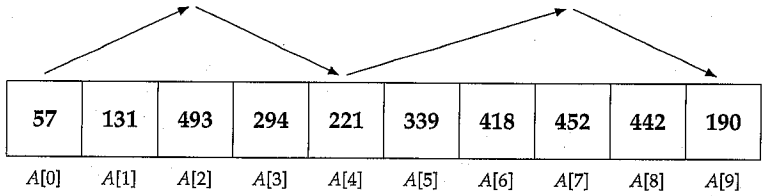


Figure 10.1: A 4-increasing-decreasing array.

Problem 10.2: Design an efficient algorithm for sorting a k-increasing-decreasing array. You are given another array of the same size that the result should be written to, and you can use $O(k)$ additional storage. *pg. 249*

10.3 Stacks and queues from heap

We discussed the notion of reduction when describing problem solving patterns. Usually, reductions are used to solve a more complex problem using a solution to a simpler problem as a subroutine.

Occasionally it makes sense to go the other way—for example, if we need the functionality of a heap, we can use a BST library, which is more commonly available, with modest performance penalties with respect, for example, to an array-based implementation of a heap.

Problem 10.3: How would you implement a stack API using a heap and a queue API using a heap? *pg. 250*

10.4 Closest stars

Consider a coordinate system for the Milky Way, in which the Earth is at $(0, 0, 0)$. Model stars as points, and assume distances are in light years. The Milky Way consists of approximately 10^{12} stars, and their coordinates are stored in a file in comma-separated values (CSV) format—one line per star and four fields per line, the first corresponding to an ID, and then three floating point numbers corresponding to the star location.

Problem 10.4: How would you compute the k stars which are closest to the Earth? You have only a few megabytes of RAM. *pg. 251*

10.5 The k-th largest element—streaming case

The goal of this problem is to design an algorithm that continuously outputs the k-th largest element in a sequence of elements that is presented one element at a time.

(For the first k cycles, the algorithm should output the smallest element.) The length of the sequence is not known in advance, and could be very large.

Problem 10.5: Design an $O(n \log k)$ time algorithm that reads a sequence of n elements and for each element, starting from the k-th element, prints the k-th largest element read up to that point. The length of the sequence is not known in advance. Your algorithm cannot use more than $O(k)$ additional storage. *pg. 253*

10.6 Approximate sort

Consider a situation where your data is almost sorted—for example, you are receiving timestamped stock quotes and earlier quotes may arrive after later quotes because of differences in server loads and network routes. What would be the most efficient way of restoring the total order?

Problem 10.6: The input consists of a very long sequence of numbers. Each number is at most k positions away from its correctly sorted position. Design an algorithm that outputs the numbers in the correct order and uses $O(k)$ storage, independent of the number of elements processed. *pg. 253*

10.7 Closest to median

Suppose you have an array A of n items, and you want to find the k items in A closest to the median of A. For example, if A contains the nine values $\langle 7, 14, 10, 12, 2, 11, 29, 3, 4 \rangle$ and $k = 5$, then the answer would be the values $\{7, 14, 10, 12, 11\}$ where the median is 10.

Problem 10.7: Design an $O(n)$ time algorithm to compute the k elements closest to the median of an array A. *pg. 254*

10.8 Online median

You want to compute the running median of a sequence of numbers. The sequence is presented to you in a streaming fashion—you cannot back up to read an earlier value, and you need to output the median after reading in each new element.

Problem 10.8: Design an algorithm for computing the running median of a sequence. The time complexity should be $O(\log n)$ per element read in, where n is the number of values read in up to that element. *pg. 255*

10.9 Generating numbers of the form $a + b\sqrt{2}$

Let S_q be the set of real numbers of the form $a + b\sqrt{q}$, where a and b are nonnegative integers, and q is an integer which is not the square of another integer. Such sets have special properties, e.g., they are closed under addition and multiplication. The first few numbers of this form are given in Figure 10.2 on the facing page.

Problem 10.9: Design an algorithm for efficiently computing the k smallest real numbers of the form $a + b\sqrt{2}$ for nonnegative integers a and b. *pg. 256*

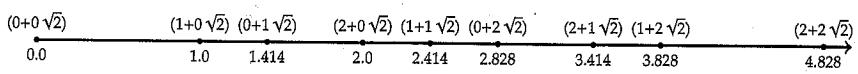


Figure 10.2: Points of the form $a + b\sqrt{2}$.

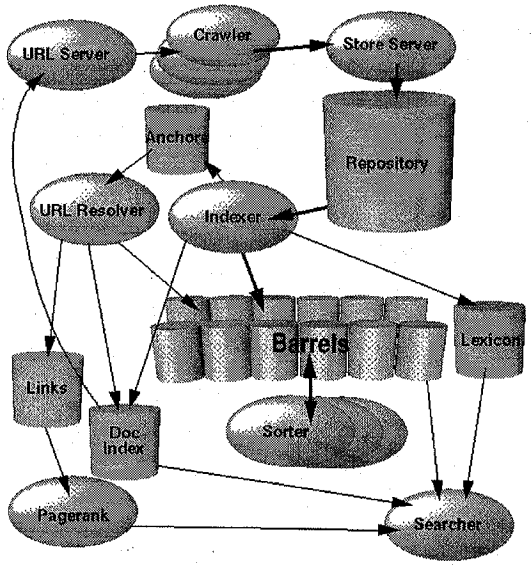
10.10 Compare with the k-th largest element

Problem 10.10: Design an $O(k)$ time algorithm for determining whether the k-th largest element in a max-heap is smaller than, equal to, or larger than a given x. The max-heap is represented using an array. Your algorithm's time complexity should be independent of the number of elements in the max-heap, and may use $O(k)$ additional storage. It cannot make any changes to the max-heap, and should handle the possibility of duplicate entries. *pg. 258*

CHAPTER

11

Searching



— *"The Anatomy of A Large-Scale Hypertextual Web Search Engine,"*
S. M. BRIN AND L. PAGE, 1998

Given an arbitrary collection of n keys, the only way to determine if a search key is present is by examining each element. This has $\Theta(n)$ time complexity. If the collection is "organized", searching can be sped up dramatically. If the data are dynamic, that is inserts and deletes are interleaved with searching, keeping the collection organized becomes more challenging.

Binary search

Binary search is at the heart of more interview questions than any other single algorithm. Fundamentally, binary search is a natural elimination-based strategy for searching a sorted array. The idea is to eliminate half the keys from consideration by keeping the keys in sorted order. If the search key is not equal to the middle element of the array, one of the two sets of keys to the left and to the right of the middle element can be eliminated from further consideration.

Questions based on binary search are ideal from the interviewers perspective: it is a basic technique that every reasonable candidate is supposed to know and it can be implemented in a few lines of code. On the other hand, binary search is much trickier to implement correctly than it appears—you should implement it as well as write corner case tests to ensure you understand it properly.

Many published implementations are incorrect in subtle and not-so-subtle ways—a study reported that it is correctly implemented in only five out of twenty textbooks. Jon Bentley, in his book *"Programming Pearls"* reported that he assigned binary search in a course for professional programmers and found that 90% failed to code it correctly despite having ample time. (Bentley's students would have been gratified to know that his own published implementation of binary search, in a column titled "Writing Correct Programs", contained a bug that remained undetected for over twenty years.)

Binary search can be written in many ways—recursive, iterative, different idioms for conditionals, etc. Here is an iterative implementation adapted from Bentley's book, which includes his bug.

```
1  int bsearch(const int &t, const vector<int> &A) {
2    int L = 0, U = A.size() - 1;
3    while (L <= U) {
4      int M = (L + U) / 2;
5      if (A[M] < t) {
6        L = M + 1;
7      } else if (A[M] == t) {
8        return M;
9      } else {
10       U = M - 1;
11     }
12   }
13   return -1;
14 }
```

The error is in the assignment `M = (L + U) / 2` in Line 4, which can lead to overflow. A common solution is to use `M = L + (U - L) / 2`.

However, even this refinement is problematic in a C-style implementation. *The C Programming Language (2nd ed.)* by Kernighan and Ritchie (Page 100) states: "If one is sure that the elements exist, it is also possible to index backwards in an array; `p[-1]`, `p[-2]`, etc. are syntactically legal, and refer to the elements that immediately precede `p[0]`." In the expression `L + (U - L) / 2`, if `U` is a sufficiently large positive integer and `L` is a sufficiently large negative integer, `(U - L)` can overflow, leading to out of bounds array access. The problem is illustrated below:

```
#define N 3000000000
char A[N];
char *B = (A + 1500000000);
int L = -1499000000;
int U = 1499000000;
// On a 32-bit machine (U - L) = -1296967296 because the actual value,
// 2998000000 is larger than 2^31 - 1. Consequently, the bsearch function
// called below sets m to -2147483648 instead of 0, which leads to an
```

```
// out-of-bounds access, since the most negative index that can be applied
// to B is -1500000000.
int result = binary_search(key, B, L, U);
```

The solution is to check the signs of L and U. If U is positive and L is negative, M = (L + U) / 2 is appropriate, otherwise set M = L + (U - L) / 2.

In our solutions that make use of binary search, L and U are nonnegative and so we use M = L + (U - L) / 2 in the associated programs.

The time complexity of binary search is given by $T(n) = T(n/2) + c$, where c is a constant. This solves to $T(n) = O(\log n)$, which is far superior to the $O(n)$ approach needed when the keys are unsorted. A disadvantage of binary search is that it requires a sorted array and sorting an array takes $O(n \log n)$ time. However if there are many searches to perform, the time taken to sort is not an issue.

Many variants of searching a sorted array require a little more thinking and create opportunities for missing corner cases.

11.1 Search a sorted array for first occurrence of k

Binary search commonly asks for the index of any element of a sorted array A that is equal to a given element. The following problem has a slight twist on this.

-14	-10	2	108	108	243	285	285	285	401
$A[0]$	$A[1]$	$A[2]$	$A[3]$	$A[4]$	$A[5]$	$A[6]$	$A[7]$	$A[8]$	$A[9]$

Figure 11.1: A sorted array with repeated elements.

Problem 11.1: Write a method that takes a sorted array A and a key k and returns the index of the *first* occurrence of k in A. Return -1 if k does not appear in A. For example, when applied to the array in Figure 11.1 your algorithm should return 3 if $k = 108$; if $k = 285$, your algorithm should return 6. *pg. 259*

11.2 Search a sorted array for the first element larger than k

Sometimes we want the first element larger than a given element.

Problem 11.2: Design an efficient algorithm that takes a sorted array A and a key k, and finds the index of the *first* occurrence an element larger than k; return -1 if every element is less than or equal to k. For example, when applied to the array in Figure 11.1 your algorithm should return -1 if $k = 500$; if $k = 101$, your algorithm should return 3. *pg. 259*

11.3 Search a sorted array for $A[i] = i$

Problem 11.3: Design an efficient algorithm that takes a sorted array A of distinct integers, and returns an index i such that $A[i] = i$ or indicate that no such index exists

by returning −1. For example, when the input is the array shown in in Figure 11.1 on the preceding page, your algorithm should return 2. *pg. 260*

11.4 Search for a pair in an abs-sorted array (☺)

An abs-sorted array is an array of numbers in which $|A[i]| \leq |A[j]|$ whenever $i < j$. For example, the array in Figure 11.2, though not sorted in the standard sense, is abs-sorted.

-49	75	103	-147	164	-197	-238	314	348	-422
$A[0]$	$A[1]$	$A[2]$	$A[3]$	$A[4]$	$A[5]$	$A[6]$	$A[7]$	$A[8]$	$A[9]$

Figure 11.2: An abs-sorted array.

Problem 11.4: Design an algorithm that takes an abs-sorted array A and a number k, and returns a pair of indices of elements in A that sum up to k. For example, if the input to your algorithm is the array in Figure 11.2 and $k = 167$, your algorithm should output $(3, 7)$. Output $(-1, -1)$ if there is no such pair. *pg. 261*

11.5 Search a cyclically sorted array

An array A of length n is said to be cyclically sorted if the smallest element in the array is at index i, and the sequence $\langle A[i], A[i+1], \ldots, A[n-1], A[0], A[1], \ldots, A[i-1] \rangle$ is sorted in increasing order, as illustrated in Figure 11.3.

378	478	550	631	103	203	220	234	279	368
$A[0]$	$A[1]$	$A[2]$	$A[3]$	$A[4]$	$A[5]$	$A[6]$	$A[7]$	$A[8]$	$A[9]$

Figure 11.3: A cyclically sorted array.

Problem 11.5: Design an $O(\log n)$ algorithm for finding the position of the smallest element in a cyclically sorted array. Assume all elements are distinct. For example, for the array in Figure 11.3, your algorithm should return 4. *pg. 263*

11.6 Search a sorted array of unknown length (☺)

Problem 11.6: Let A be a sorted array. The length of A is not known in advance; accessing $A[i]$ for i beyond the end of the array throws an exception. Design an algorithm that takes A and a key k and returns an index i such that $A[i] = k$; return -1 if k does not appear in A. *pg. 264*

11.7 Completion search

You are working in the finance office for ABC corporation. The total payroll expense last year was $\$S$. This year, the corporation needs to cut payroll expenses to $\$S'$. The chief executive officer wants to put a cap σ on salaries. Every employee who earned more than $\$\sigma$ last year will be paid $\$\sigma$ this year; employees who earned less than $\$\sigma$ will see no change in their salary.

For example, given five employees with salaries $\$90, \$30, \$100, \$40$, and $\$20$, and $S' = 210$, then 60 is a suitable value for σ.

Problem 11.7: Let A be an array of n nonnegative real numbers and S' be a nonnegative real number less than $\sum_{i=0}^{n-1} A[i]$. Design an efficient algorithm for computing σ such that $\sum_{i=0}^{n-1} \min(A[i], \sigma) = S'$, if such a σ exists. *pg. 265*

11.8 Searching in two sorted arrays

The k-th smallest element in a sorted array A is simply $A[k-1]$ which takes $O(1)$ time to compute. Suppose you are given two sorted arrays A and B, of length n and m respectively, and you need to find the k-th smallest element of the array C consisting of the $n+m$ elements of A and B arranged in sorted order. (We'll refer to this array as the union of A and B, although strictly speaking union is a set-theoretic operation that does not have a notion of order, or duplicate elements.)

You could merge the two arrays into a third sorted array and then look for the answer, but the merge would take $O(n+m)$ time. You can build the merged array on the first k elements, which would be an $O(k)$ operation.

Problem 11.8: You are given two sorted arrays A and B of lengths m and n, respectively, and a positive integer $k \in [1, m+n]$. Design an algorithm that runs in $O(\log k)$ time for computing the k-th smallest element in array formed by merging A and B. Array elements may be duplicated within and between A and B. *pg. 266*

11.9 Computing square roots

Square root computations can be implemented using sophisticated numerical techniques involving iterative methods and logarithms. However if you were asked to implement a square root function, you would not be expected to know these techniques.

Problem 11.9: Implement a function which takes as input a floating point variable x and returns $\sqrt{x}$. *pg. 267*

11.10 2D array search

Problem 11.10: Let A be an $n \times n$ 2D array where rows and columns are sorted in increasing sorted order. Design an efficient algorithm that decides whether a number x appears in A. How many entries of A does your algorithm inspect in the worst-case? Can you prove a tight lower bound that any such algorithm has to consider in the worst-case? *pg. 268*

11.11 Finding the winner and runner-up

One hundred and twenty eight players take part in a tennis tournament. The "x beats y" relationship is transitive, i.e., for all players a, b, and c, if a beats b and b beats c, then a beats c.

Problem 11.11: How would you organize a tournament with 128 players to minimize the number of matches needed to find the best player? How many matches do you need to find the best and the second best player? *pg. 269*

Searching unsorted arrays

Now we consider a number of problems related to searching arrays that are not sorted, implying that we cannot use elimination. The problems in this section can be solved without sorting, and the solutions have $O(n)$ time complexity, where n is the length of the array. We study similar problems in Chapter 13, but for those problems, the best solutions entail sorting.

11.12 Finding the min and max simultaneously

Given an array of n objects that are comparable, you can find either the *min* or the *max* of the elements in the array with $n-1$ comparisons. Comparing elements may be expensive, e.g., a comparison may involve a number of nested calls or the elements being compared may be long strings. Therefor is is natural to ask if both the min and the max of an array can be computed with less than the $2n-3$ comparisons required to compute the min and the max independently.

Problem 11.12: Find the min and max elements from an array of n elements using no more than $\lceil 3n/2 \rceil - 2$ comparisons. *pg. 270*

11.13 The k-th largest element

Let A be an array of length n. Assume that that entries are distinct, i.e., if $i \neq j$ then $A[i] \neq A[j]$. The array B is said to be a descending sorting of A if $|B| = |A|$, there exists a permutation σ of $\{0, 1, 2, \ldots, n-1\}$ such that $A[i] = B[\sigma(i)]$, and B is sorted in descending order. The k-th largest element of A is defined to be the k-th element of B.

The k-th order statistic of a collection is its k-th smallest value, with the minimum element being the first order statistic. In this parlance, we are asking for the $k+1$-th order statistic of the collection A.

Problem 11.13: Design an algorithm for computing the k-th largest element in an array A that runs in $O(n)$ expected time. *pg. 270*

11.14 The k-th largest element—large n and small k (☹)

The goal of this problem is to design an algorithm for computing the k-th largest element in a sequence of elements that is presented one element at a time. The length of the sequence is not known in advance, and could be very large.

Problem 11.14: Design an algorithm for computing the k-th largest element in a sequence of elements. It should run in $O(n)$ expected time where n is the length of the sequence, which is not known in advance. The value k is known in advance. Your algorithm should print the k-th largest element after the sequence has ended. It should use $O(k)$ additional storage. *pg. 271*

11.15 Finding a missing element

The storage capacity of hard drives dwarfs that of RAM. This can lead to interesting space-time trade-offs.

Problem 11.15: Suppose you were given a file containing roughly one billion Internet Protocol (IP) addresses, each of which is a 32-bit unsigned integer. How would you programmatically find an IP address that is not in the file? Assume you have unlimited drive space but only two megabytes of RAM at your disposal. *pg. 272*

11.16 Find the duplicate and missing elements

Let A be an array containing $n-1$ integers, each in the set $\mathcal{Z}_n = \{0, 1, \ldots, n-1\}$. Suppose exactly one element $m \in \mathcal{Z}_n$ is not present in A. We can determine m in $O(n)$ time and $O(1)$ space by computing $\text{Sum}(A)$, the sum of the elements in A. The sum of all the elements in $\mathcal{Z}_n$ is $\text{Sum}(\mathcal{Z}_n) = \frac{n(n-1)}{2}$. Hence $\text{Sum}(\mathcal{Z}_n) - \text{Sum}(A)$ equals the missing element m. Similarly, if A contains $n+1$ elements drawn from the set $\mathcal{Z}_n$, with exactly one element t appearing twice in A, the element t will be equal to $\text{Sum}(A) - \text{Sum}(\mathcal{Z}_n)$.

Alternately, for the first problem, we can compute m by computing the XOR of all the elements in $\mathcal{Z}_n$ and XORing that with the XOR of all the elements in A—every element in A, except for the missing element, cancels out since it is also present in $\mathcal{Z}_n$. Therefore the resulting XOR equals m. The same approach works for the second problem.

Problem 11.16: Let A be an array of n integers in $\mathcal{Z}_n$, with exactly one element t appearing twice. This implies exactly one element $m \in \mathcal{Z}_n$ is missing from A. How would you compute t and m in $O(n)$ time and $O(1)$ space? *pg. 273*

11.17 Find the element that appears only once

Given an integer array where each element appears twice except for one that appears only once, we can use $O(n)$ space and $O(n)$ time to find the element that appears exactly once, e.g., using a hash table. However, there is a better solution: compute the bitwise-XOR ($\oplus$) of each element of the array. Because $x \oplus x = 0$, all elements that appear an even number of times cancel out, and the element that appears remains. Therefore, this problem can be solved using $O(1)$ space.

Problem 11.17: Given an array A, in which each element of A appears three times except for one element e that appears once, find e in $O(1)$ space and $O(n)$ time. *pg. 274*

11.18 Searching an array with close entries

An array of integers A is said to be *close* if for each $i \in [0, |A| - 2]$, $|A[i] - A[i+1]| \leq 1$, e.g., as in Figure 11.4.

-1	0	0	1	2	2	1	2	3	4
$A[0]$	$A[1]$	$A[2]$	$A[3]$	$A[4]$	$A[5]$	$A[6]$	$A[7]$	$A[8]$	$A[9]$

Figure 11.4: A close array.

Problem 11.18: Design an efficient algorithm that takes a close array A, and a key k and searches for any index j such that $A[j] = k$. Return -1 if no such index exists. For example, for the array in Figure 11.4, if $k = 2$, your algorithm should return an index in $\{4, 5, 7\}$. *pg. 275*

11.19 Majority find

Several applications require identification of tokens—objects which implement an equals method—in a sequence which appear more than a specified fraction of the total number of tokens. For example, we may want to identify the users using the largest fraction of the network bandwidth or IP addresses originating the most Hypertext Transfer Protocol (HTTP) requests. Here we consider a simplified version of this problem.

Problem 11.19: You are reading a sequence of words from a very long stream. You know *a priori* that more than half the words are repetitions of a single word w (the "majority element") but the positions where w occurs are unknown. Design an algorithm that makes a single pass over the stream and uses only a constant amount of memory to identify w. *pg. 275*

CHAPTER

12

Hash Tables

The new methods are intended to reduce the amount of space required to contain the hash-coded information from that associated with conventional methods. The reduction in space is accomplished by exploiting the possibility that a small fraction of errors of commission may be tolerable in some applications.

— *"Space/time trade-offs in hash coding with allowable errors,"*
B. H. BLOOM, 1970

The idea underlying a *hash table* is to store objects according to their key field in an array. Objects are stored in array locations based on the "hash code" of the key. The hash code is an integer computed from the key by a hash function. If the hash function is chosen well, the objects are distributed uniformly across the array locations.

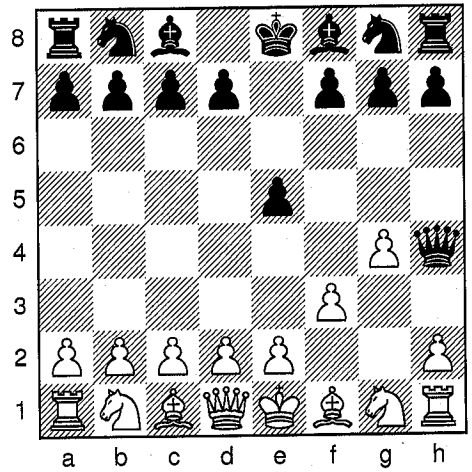
If two keys map to the same location, a "collision" is said to occur. The standard mechanism to deal with collisions is to maintain a linked list of objects at each array location. If the hash function does a good job of spreading objects across the underlying array and take $O(1)$ time to compute, on average, lookups, insertions, and deletions have $O(1 + n/m)$ time complexity, where n is the number of objects and m is the length of the array. If the "load" n/m grows large, rehashing can be applied to the hash table. A new array with a larger number of locations is allocated, and the objects are moved to the new array. Rehashing is expensive ($\Theta(n + m)$ time) but if it is done infrequently (for example, whenever the number of entries doubles), its amortized cost is low.

A hash table is qualitatively different from a sorted array—keys do not have to appear in order, and randomization (specifically, the hash function) plays a central role. Compared to binary search trees (discussed in Chapter 14), inserting and deleting in a hash table is more efficient (assuming rehashing is infrequent). One disadvantage of hash tables is the need for a good hash function but this is rarely an issue in practice. Similarly, rehashing is not a problem outside of realtime systems and even for such systems, a separate thread can do the rehashing.

12.1 DESIGN A HASH FUNCTION FOR DICTIONARIES

A hash function has one hard requirement—two keys that are identical should yield the same hash code. This may seem obvious, but is easy to get wrong, e.g., by writing a hash function that is based on address rather than contents.

Problem 12.1: Design a hash function that is suitable for words in a dictionary.
pg. 276



1. f3, e5 2. g4, ♕h4

Figure 12.1: Chessboard corresponding to the fastest checkmate, *Fool's Mate.*

12.2 A hash function for the state of a chess game

The state of a game of chess is determined by what piece is present on each square, as illustrated in Figure 12.1. Each square may be empty, or have one of six classes of pieces; each piece may be black or white. Thus $\lceil \lg(1 + 6 \times 2) \rceil = 4$ bits suffice per square, which means that a total of $64 \times 4 = 256$ bits can represent the state of the chessboard. (The true state is somewhat more complex, as it needs to capture which side is to move, castling rights, *en passant*, etc.)

Chess playing computers need to store sets of states, e.g., to determine if a particular state has been evaluated before, or is known to be a winning state. To reduce storage, it is natural to apply a hash function to the 256 bits of state, and ignore collisions. The hash code can be computed by a conventional hash function for strings. However, since the computer repeatedly explores nearby states, it is advantageous to consider hash functions that can be efficiently computed based on incremental changes to the board.

Problem 12.2: Design a hash function for chess game states. Your function should take a state and the hash code for that state, and a move, and efficiently compute the hash code for the updated state. *pg. 277*

12.3 Nearest repetition

People do not like reading text in which a word is used multiple times in a short paragraph. You are to write a function which helps identify such a problem.

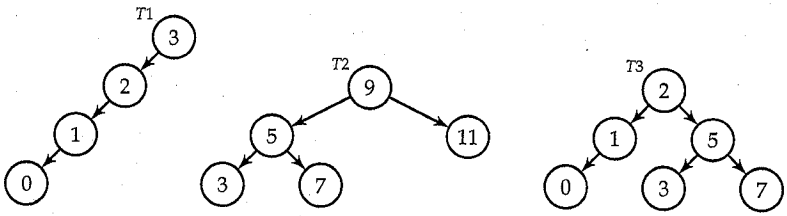
Problem 12.3: Let s be an array of strings. Write a function which finds a closest pair of equal entries. For example, if s = ["All", "work", "and", "no", "play", "makes",

"for", "no", "work", "no", "fun", "and", "no", "results"], then the second and third occurrences of "no" is the closest pair. pg. 277

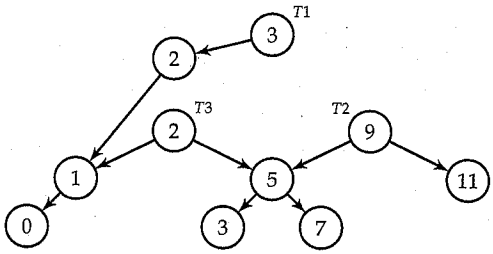
12.4 Binary tree compression

Suppose you have an application which will use a very large number of binary trees. You know that many of the subtrees will be identical, and you want to avoid duplicating the storage required for these subtrees by sharing identical subtrees.

Formally, binary trees A and B are isomorphic if both are `null`, or their roots store the same key and $A.left$ is isomorphic to $B.left$ and $A.right$ is isomorphic to $B.right$. Since each node in a subtree is the root of a binary tree, the notion of isomorphism generalizes to nodes.



(a) Three binary trees, $T1$, $T2$, and $T3$, duplicate isomorphic nodes.



(b) After canonicalization.

Figure 12.2: Binary tree canonicalization for $T1$, $T2$, and $T3$.

Problem 12.4: Given a set of binary trees $A_1, \ldots, A_n$ how would you compute a new set of binary trees $B_1, \ldots, B_n$ such that for each i, $1 \leq i \leq n$, A_i and B_i are isomorphic, and no pair of isomorphic nodes exists in the set of nodes defined by $B_1, \ldots, B_n$. (This is sometimes referred to as the canonical form.) Assume nodes are not shared in $A_1, \ldots, A_n$. See Figure 12.2 for an example. pg. 278

12.5 Pair users by attributes

You are building a social network site where each user specifies a set of attributes. You would like to pair each user with another unpaired user that specified exactly the same set of attributes.

Problem 12.5: You are given a sequence of users where each user has a unique 32-bit integer key and a set of attributes specified as strings. When you read a user, you should pair that user with another previously read user with identical attributes who is currently unpaired, if such a user exists. If the user cannot be paired, you should keep him in the unpaired set. How would you implement this matching process efficiently? *pg. 279*

12.6 Pair users by attributes, approximate matching (☹)

Problem 12.6: Solve Problem 12.5 on the facing page when users are grouped based on having similar attributes. The similarity between two sets of attributes A and B is $\frac{|A \cap B|}{|A \cup B|}$. *pg. 279*

12.7 Anagrams

Anagrams are popular word play puzzles, where by rearranging letters of one set of words, you get another set of words. For example, "eleven plus two" is an anagram for "twelve plus one". Crossword puzzle enthusiasts would like to be able to generate all possible anagrams for a given set of letters.

Problem 12.7: Write a function that takes as input a dictionary of English words, and returns a partition of the dictionary into subsets of words that are all anagrams of each other. *pg. 280*

12.8 Can a string be permuted to form a palindrome?

A palindrome is a word that reads the same forwards and backwards, e.g., "level" and "rotator".

Problem 12.8: Write a program to test whether the letters forming a string s can be permuted to form a palindrome. For example, "edified" can be permuted to form "deified". Explore solutions that trade time for space. *pg. 281*

12.9 Anonymous letter

A hash table can be viewed as a dictionary. For this reason, hash tables commonly appear in string processing.

Problem 12.9: You are required to write a method which takes an anonymous letter L and text from a magazine M. Your method is to return `true` iff L can be written using M, i.e., if a letter appears k times in L, it must appear at least k times in M. *pg. 282*

12.10 Line through the most points (☹)

Problem 12.10: Let P be a set of n points in the plane. Each point has integer coordinates. Design an efficient algorithm for computing a line that contains the maximum number of points in P. *pg. 282*

12.11 Search for frequent items

This problem is a continuation of Problem 11.19 on Page 91. In practice we may not be interested in just the majority token but all the tokens whose count exceeds say 1% of the total token count. It is simple to show that it is impossible to do this in a single pass when you have limited memory but if you are allowed to pass through the sequence twice, it is possible to identify the common tokens.

Problem 12.11: You are reading a sequence of strings separated by white space. You are allowed to read the sequence twice. Devise an algorithm that uses $O(k)$ memory to identify the words that occur at least $\lceil \frac{n}{k} \rceil$ times, where n is the length of the sequence. *pg. 284*

12.12 Automatic hyphenation

To allow efficient usage of paper, and regular appearance of right-side margins, words may be divided and a hyphen inserted to indicate that the letters form a word fragment, not a word.

Words cannot be divided arbitrarily. A small set of rules can be applied to most words to determine where they can be hyphenated; however there are many words to which they cannot be applied.

Let R be the set of words that can be split using rules and E the words that cannot. The set E is large enough that it must be stored on disk; R is larger still. Since the majority of words are not in E, it is advantageous to determine if a word is not in E without going to disk.

Problem 12.12: Design a scheme for checking membership in E that minimizes the number of disk accesses. Assume that $|R| = 10^6, |E| = 10^5$, and you can use up to 1.25×10^5 bytes of RAM. *pg. 286*

12.13 Plagiarism detector

Problem 12.13: A pair of strings is *k-suspicious* if they have a substring of length greater than or equal to k in common. Design an efficient algorithm that takes as input a set of strings and positive integer k, and returns all pairs of strings that are k-suspicious. Assume that most pairs will not be k-suspicious. *pg. 286*

12.14 Smallest subarray covering set

When you type keywords in a search engine, the search engine will return results, and each result contains a digest of the web page, i.e., a highlighting within that page of the keywords that you searched for. For example, a search for the keywords "Union" and "save" on a page with the text of the Emancipation Proclamation should return the result shown in Figure 12.3 on the facing page.

The digest for this page is the text in boldface, with the keywords underlined for emphasis. It is the shortest substring of the page which contains all the keywords in the search. The problem of computing the digest is abstracted as follows.

> My paramount object in this struggle is to **save the Union**, and is not either to save or to destroy slavery. If I could save the Union without freeing any slave I would do it, and if I could save it by freeing all the slaves I would do it; and if I could save it by freeing some and leaving others alone I would also do that.

Figure 12.3: Search result with digest in boldface and search keywords underlined.

Problem 12.14: Let A and Q be arrays of strings. Define the subarray $A[i:j]$ to *cover* Q if for all $k \in [0, |Q|-1]$, there exists $l \in [i, j]$, $Q[k] = A[l]$. Write a function that takes two arrays A and Q and computes a minimum length subarray $A[i:j]$ that covers Q.

Suppose that A is presented in streaming fashion, i.e., elements are read one at a time, and you cannot read earlier entries. The array Q is much smaller, and can be stored in RAM. How would you modify your solution for this case? *pg. 287*

12.15 Smallest subarray sequentially covering set (⚙)

In Problem 12.14 on the preceding page we did not differentiate between the order in which keywords appeared. If the digest has to include the keywords in the order in which they appear in the search textbox, we may get a different digest. For example, for the search keywords "Union" and "save", in that order, the digest would be "**Union, and is not either to save**".

Building on Problem 12.14 on the facing page, define the subarray $A[i:j]$ to sequentially cover Q iff there exist $k_0, k_1, \ldots, k_{|Q|-1}$ such that $i = k_0 < k_1 < \cdots < k_{|Q|-1} = j$ and for all $l \in [0, |Q|-1]$, $Q[l] = A[k_l]$.

Problem 12.15: Write a function that takes two integer-valued arrays A and Q and computes a minimum length subarray $A[i:j]$ that sequentially covers Q. Assume all elements in Q are distinct. *pg. 289*

12.16 ISBN cache

The International Standard Book Number (ISBN) is a unique commercial book identifier based on the 9-digit standard book numbering code. The 10-digit ISBN was ratified by the International Organization for Standardization (ISO) in 1974; since 2007, ISBNs have contained 13 digits. The last digit in a 10-digit ISBN is the check digit—it is the sum of the first 9 digits, modulo 11; a 10 is represented by an "X". For 13 digit ISBNs, the last digit is also a check digit but is guaranteed to be between 0 and 9.

Problem 12.16: Implement a cache for looking up prices of books identified by their ISBN. Use the Least Recently Used (LRU) strategy for cache eviction policy. *pg. 290*

CHAPTER

13 Sorting

A description is given of a new method of sorting in the random-access store of a computer. The method compares favorably with other known methods in speed, in economy of storage, and in ease of programming.

— "Quicksort,"
C. A. R. HOARE, 1962

Sorting—rearranging a collection of items into increasing or decreasing order—is a common problem in computing. Sorting is used to preprocess the collection to make searching faster (as we saw with binary search through an array), as well as identify items that are similar (e.g., students are sorted on test scores).

Naïve sorting algorithms run in $\Theta(n^2)$ time. A number of sorting algorithms run in $O(n \log n)$ time—heapsort, merge sort, and quicksort are examples. Each has its advantages and disadvantages: for example, heapsort is in-place but not stable; merge sort is stable but not in-place; quicksort runs $O(n^2)$ time in worst case. (An in-place sort is one which uses $O(1)$ space; a stable sort is one where entries which are equal appear in their original order.) Most sorting routines are based on a compare function that takes two items as input and returns -1 if the first item is smaller than the second item, 0 if they are equal and 1 otherwise. However it is also possible to use numerical attributes directly, e.g., in radix sort.

The heap data structure is discussed in detail in Chapter 10. Briefly, a max-heap (min-heap) stores keys drawn from an ordered set. It supports $O(\log n)$ inserts and $O(1)$ time lookup for the maximum (minimum) element; the maximum (minimum) key can be deleted in $O(\log n)$ time. Heaps can be helpful in sorting problems, as illustrated by Problems 10.1 on Page 80, 10.2 on Page 81, and 10.6 on Page 82.

13.1 GOOD SORTING ALGORITHMS

Problem 13.1: What is the most efficient sorting algorithm for each of the following situations:

- A large array whose entries are random numbers.
- A small array of numbers.
- A large array of numbers that is already almost sorted.
- A large collection of integers that are drawn from a small range.
- A large collection of numbers most of which are duplicates.

– Stability is required, i.e., the relative order of two records that have the same sorting key should not be changed.

pg. 292

13.2 Variable length sort

Most sorting algorithms rely on a basic swap step. When records are of different lengths, the swap step becomes nontrivial.

Problem 13.2: Sort lines of a text file that has one million lines such that the average length of a line is 100 characters but the longest line is one million characters long.

pg. 292

13.3 Least distance sorting

You come across a collection of 20 stone statues in a line. You want to sort them by height, with the shortest statue on the left. The statues are heavy and you want to move them the least possible distance.

Problem 13.3: Design a sorting algorithm that minimizes the total distance that items are moved.

pg. 293

13.4 Counting sort

Suppose you need to reorder the elements of a very large array so that equal elements appear together. More formally, if A is an array, you are to permute the elements of A so that after the permutation $\forall i < j < k\ A[i] = A[k] \Rightarrow A[j] = A[i]$.

If the entries are integers, this can be done by sorting the array. If the number of distinct integers is very small relative to the size of the array, an efficient approach to sorting the array is to count the number of occurrences of each distinct integer and write the appropriate number of each integer, in sorted order, to the array.

Problem 13.4: You are given an array of n `Person` objects. Each `Person` object has a field `key`. Rearrange the elements of the array so that `Person` objects with equal keys appear together. The order in which distinct keys appear is not important. Your algorithm must run in $O(n)$ time and $O(k)$ additional space. How would your solution change if keys have to appear in sorted order?

pg. 293

13.5 Intersect two sorted arrays

A natural implementation for a search engine is to retrieve documents that match the set of words in a query by maintaining an inverted index. Each page is assigned an integer identifier, its *document-ID*. An inverted index is a mapping that takes a word w and returns a sorted array of page-ids which contain w—the sort order could be, for example, the page rank in descending order. When a query contains multiple words, the search engine finds the sorted array for each word and then computes the intersection of these arrays—these are the pages containing all the words in the query.

The most computationally intensive step of doing this is finding the intersection of the sorted arrays.

Problem 13.5: Given sorted arrays A and B of lengths n and m respectively, return an array C containing elements common to A and B. The array C should be free of duplicates. How would you perform this intersection if—(1.) $n \approx m$ and (2.) $n \ll m$?
pg. 295

13.6 Team photo day—1

You are a photographer for a soccer meet. You will be taking pictures of pairs of opposing teams. All teams have the same number of players. A team photo consist of a front row of players and a back row of players. A player in the back row must be taller than the player in front of him, as illustrated in Figure 13.1. All players in a row must be from the same team.

Figure 13.1: A team photo. Each team has 10 players, and each player in the back row is taller than the corresponding player in the front row.

Problem 13.6: Design an algorithm that takes as input two teams and the heights of the players in the teams and checks if it is possible to place players to take the photo subject to the placement constraint.
pg. 296

13.7 Count the occurrences of characters in a sentence

Computers are ideally suited to taking a large amount of data and summarizing it, e.g., as gross statistics.

Problem 13.7: Given a string s, print in alphabetical order each character that appears in s, and the number of times that it appears. For example, if s ="bcdacebe", output "(a, 1), (b, 2), (c, 2), (d, 1), (e, 2)".
pg. 297

13.8 Unique elements

Suppose you are given a set of names and your job is to produce a set of unique first names. If you just remove the last name from all the names, you may have some duplicate first names. Creating a set of first names that has each name occurring only once amounts to the following.

Problem 13.8: Design an efficient algorithm for removing all the duplicates from an array.
pg. 298

13.9 Task assignment (☺)

We consider the problem of scheduling $n = 2m$ tasks to be performed by m workers. Each worker must be assigned exactly two tasks. Each task has a duration. Tasks are independent, i.e., there are no constraints of the form "Task 4 cannot start before Task 3 is completed." We want to assign tasks to workers so as to minimize how long it takes before all tasks are completed.

Formally, let A be an array of positive numbers of length $n = 2m$, i.e., n is even, where $A[i]$ represents the duration of Task i. Define a 2-partition Π of A to be a partition Π of $\{0, 1, \ldots, n-1\}$ into $\frac{n}{2}$ subsets, $P_0, P_1, \ldots, P_{\frac{n}{2}-1}$ each with two elements. Define $Q(\Pi)$ to be $\max_{i=0}^{\frac{n}{2}-1} \left(\sum_{e \in P_i} A[e]\right)$.

Problem 13.9: Design an efficient algorithm that takes as input an array A of even length and computes a 2-partition of A that has minimum $Q(\Pi)$. *pg. 298*

13.10 Rendering a calendar

Consider the problem of designing an online calendaring application. One component of the design is to render the calendar, i.e., display it visually.

Suppose each day consists of a number of events, where an event is specified as a start time and a finish time. Individual events for a day are to be rendered as non-overlapping rectangular regions whose sides are parallel to the x- and y-axes. Let the x-axis correspond to time. If an event starts at time b and ends at time e, the upper and lower sides of its corresponding rectangle must be at b and e, respectively. Figure 13.2 represents a set of events.

Suppose the y-coordinates for each day's events must lie between 0 and L (a pre-specified constant), and the rectangle for each event has the same "height", which is the distance between the sides parallel to the x-axis is fixed. Your task is to compute the maximum height an event rectangle can have. In essence, this is equivalent to the following problem.

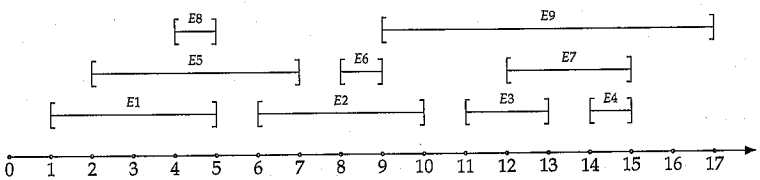


Figure 13.2: A set of nine events. The earliest starting event begins at time 1; the latest ending event ends at time 17. The maximum number of concurrent events is 3, e.g., $\{E1, E5, E8\}$ as well as others.

Problem 13.10: Given a set of events, how would you determine the maximum number of events that take place concurrently? *pg. 299*

13.11 Union of intervals

In this problem we consider sets of intervals with integer endpoints; the intervals may be open or closed at either end. We want to compute the union of the intervals in such sets. A concrete example is given in Figure 13.3.

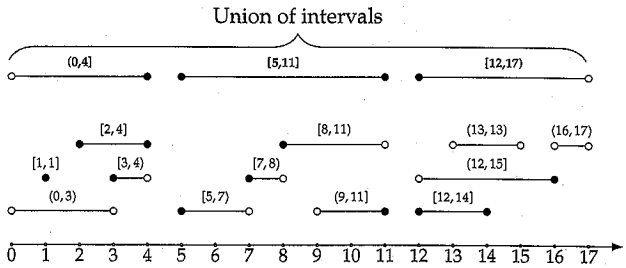


Figure 13.3: A set of intervals and their union.

Problem 13.11: Design an algorithm that takes as input a set of intervals I, and outputs the union of the intervals. What is the time complexity of your algorithm as a function of the number of intervals? *pg. 300*

13.12 Points covering intervals

Consider an engineer responsible for a number of tasks on the factory floor. Each task starts at a fixed time and ends at a fixed time. The engineer wants to visit the floor to check on the tasks. Your job is to help him minimize the number of visits he makes. In each visit, he can check on all the tasks taking place at the time of the visit. A visit takes place at a fixed time, and he can only check on tasks taking place at exactly that time.

Problem 13.12: You are given a set of n tasks modeled as closed intervals $[a_i, b_i]$, for $i = 0, \ldots, n-1$. A set S of visit times *covers* the tasks if $[a_i, b_i] \cap S \neq \emptyset$, for $i = 0, \ldots, n-1$. Design an efficient algorithm for finding a minimum cardinality set of visit times that covers all the tasks. *pg. 302*

13.13 Rays covering arcs

Let's say you are responsible for the security of a castle. The castle has a circular perimeter. A total of n robots patrol the perimeter—each robot is responsible for a closed connected subset of the perimeter, i.e., an arc. (The arcs for different robots may overlap.) You want to monitor the robots by installing cameras at the center of the castle that look out to the perimeter. Each camera can look along a ray. To save cost, you would like to minimize the number of cameras. See Figure 13.4 on the next page for an example.

Problem 13.13: Let $[\theta_i, \phi_i]$, for $i = 0, \ldots, n-1$ be n arcs, where the i-th arc is the set of points on the perimeter of the unit circle that subtend an angle in the interval $[\theta_i, \phi_i]$

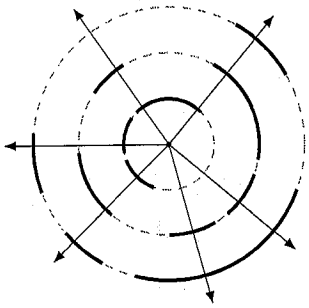


Figure 13.4: An instance of the minimum ray covering problem, with 12 partially overlapping arcs. Arcs have been drawn at different distances for illustration. For this instance, six cameras are sufficient, corresponding to the six rays.

at the center. A ray is a set of points that all subtend the same angle to the origin, and is identified by the angle they make relative to the x-axis. A set R of rays "covers" the arcs if $[\theta_i, \phi_i] \cap R \neq \emptyset$, for $i = 0, \ldots, n-1$. Design an efficient algorithm for finding a minimum cardinality set of rays that covers all arcs. *pg. 303*

13.14 The 3-sum problem (☹)

Let A be an array of n numbers. Let t be a number, and k be an integer in $[1, n]$. Define A to k-create t iff there exists k indices $i_0, i_1, \ldots, i_{k-1}$ (not necessarily distinct) such that $\sum_{j=0}^{k-1} A[i_j] = t$.

Problem 13.14: Design an algorithm that takes as input an array A and a number t, and determines if A 3-creates t. *pg. 303*

13.15 Pancake sorting (☹)

Suppose you are given an array A of n integers. "Flipping" A at k reverses the subarray $A[k : n-1]$. Suppose the only way you can move elements in an array is by flipping. (This restriction is appropriate, for example, when sorting a stack of pancakes of different sizes on a griddle by repeatedly inserting a spatula at appropriate locations and flipping, as shown in Figure 13.5.)

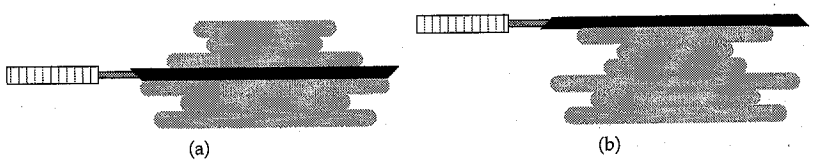


Figure 13.5: Example of a flip. The spatula is inserted in the middle of the stack of pancakes, as shown on the left, and the flip results in the configuration on the right.

Problem 13.15: Develop an algorithm for computing a short sequence of flips that will sort an array A. *pg. 304*

CHAPTER

14

Binary Search Trees

The number of trees which can be formed with $n+1$ given knots $\alpha, \beta, \gamma, \ldots = (n+1)^{n-1}$.

— *"A Theorem on Trees,"*
A. CAYLEY, 1889

Adding and deleting elements to an array is computationally expensive, particularly when the array needs to stay sorted. BSTs are similar to arrays in that the keys are in a sorted order. However, unlike arrays, elements can be added to and deleted from a BST efficiently. BSTs require more space than arrays since each node stores two pointers, one for each child, in addition to the key.

A BST is a binary tree as defined in Chapter 9 in which the nodes store keys drawn from a totally ordered set. The keys stored at nodes have to respect the BST property—the key stored at a node is greater than or equal to the keys stored at the nodes of its left subtree and less than or equal to the keys stored in the nodes of its right subtree. Figure 14.1 on the next page shows a BST whose keys are the first 16 prime numbers.

Key lookup, insertion, and deletion take time proportional to the height of the tree, which can in worst-case be $\Theta(n)$, if insertions and deletions are naïvely implemented. However there are implementations of insert and delete which guarantee the tree has height $\Theta(\log n)$. These require storing and updating additional data at the tree nodes. Red-black trees are an example of balanced BSTs and are widely used in data structure libraries, e.g., to implement maps in the Standard Template Library (STL).

The BST prototype in C++ is listed as follows:

```
template <typename T>
class BinarySearchTree {
  public:
    T data;
    shared_ptr<BinarySearchTree<T>> left, right;
};
```

14.1 DOES A BINARY TREE SATISFY THE BST PROPERTY?

Problem 14.1: Write a function that takes as input the root of a binary tree whose nodes have a key field, and returns `true` iff the tree satisfies the BST property. ***pg. 305***

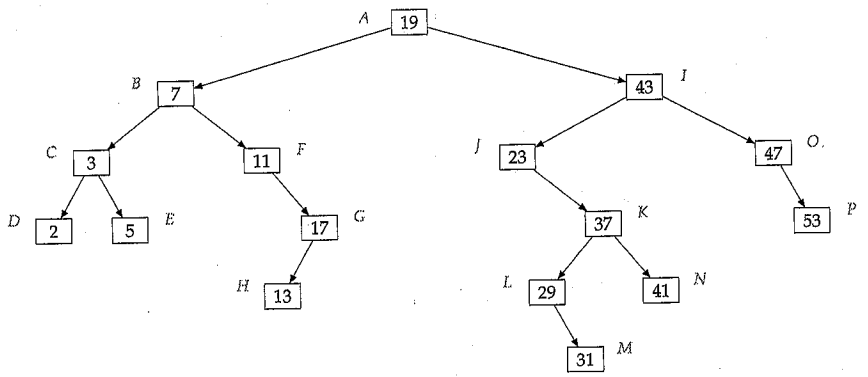


Figure 14.1: An example BST.

14.2 Successor in a BST

The successor of a node n in a BST is the node s that appears immediately after n in an inorder walk. When all keys are distinct, s holds the smallest key larger than the key at n. (The last node in the inorder walk has no successor.) For example, in Figure 14.1, the successor of Node G (with key 17) is Node A (with key 19).

Problem 14.2: Given a node x, find the successor of x in a BST. Assume that nodes have `parent` fields, and the `parent` field of root points to `null`. *pg. 308*

14.3 Updating a BST

A BST is a dynamic data structure—in particular, it supports efficient insertions and deletions of keys.

Problem 14.3: Design efficient functions for inserting and removing keys in a BST. Assume that all elements in the BST are unique, and that your insertion method must preserve this property. You cannot change the contents of any node. What are the time complexities of your functions? *pg. 308*

14.4 Search a BST for first occurrence of k

Searching for a key in a BST is very similar to binary search in a sorted array. Many variants of the basic search problem can be posed for BSTs.

Problem 14.4: Given a BST T, write recursive and iterative versions of a function that takes a BST T, a key k, and returns the node containing k that would appear first in an inorder walk. If k is absent, return `null`. For example, when applied to the BST in Figure 14.2 on the next page, your algorithm should return Node B if $k = 108$, Node G if $k = 285$, and `null` if $k = 143$. *pg. 311*

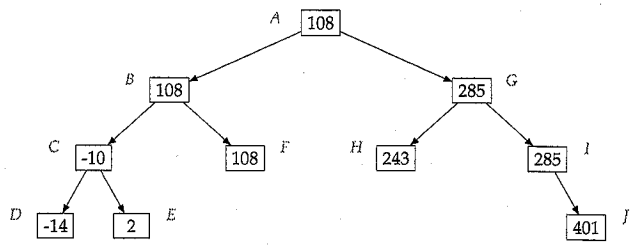


Figure 14.2: A BST with duplicate keys.

14.5 Search BST for the first key larger than k

BSTs offer more than the ability to search for a key—they can be used to find the *min* and *max* elements, look for the successor or predecessor of a given search key (which may or may not be presented in the BST), and enumerate the elements in sorted order.

Problem 14.5: Write a function that takes a BST T and a key k, and returns the first entry larger than k that would appear in an inorder walk. If k is absent or no key larger than k is present, return `null`. For example, when applied to the BST in Figure 14.1 on the preceding page you should return 29 if $k = 23$; if $k = 32$, you should return `null`. *pg. 312*

14.6 Min-first BST

A *min-first* BST is one in which the minimum key is stored at the root; each key in the left subtree is less than every key in the right subtree. The subtrees themselves are min-first BSTs. See Figure 14.3 for an example.

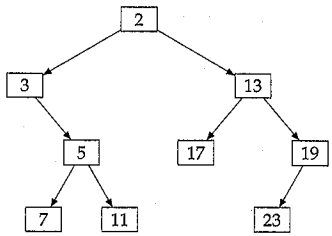


Figure 14.3: A min-first BST.

Problem 14.6: Write a function that takes a min-first BST T and a key k, and returns true iff T contains k. *pg. 313*

14.7 Building a BST from a sorted array

Let A be a sorted array of n numbers. A super-exponential number of BSTs can be built on the elements of A: $\frac{1}{n+1}\binom{2n}{n}$ to be precise. Some of these trees are skewed, and are closer to lists; others are more balanced.

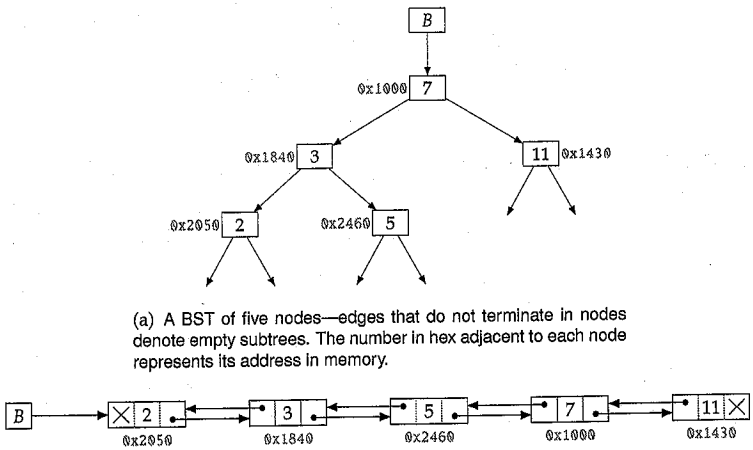
Problem 14.7: How would you build a BST of minimum possible height from a sorted array A? *pg. 314*

14.8 Build a BST from a sorted linked list

Problem 14.8: Let L be a singly linked list of numbers, sorted in ascending order. Design an efficient algorithm that takes as input L, and builds a height-balanced BST on the entries in L. Your algorithm should run in $O(n)$ time, where n is the number of nodes in L. You cannot use dynamic memory allocation—reuse the nodes of L for the BST. You can update pointer fields, but cannot change node contents. *pg. 314*

14.9 Convert a BST to sorted doubly linked list

A BST has two pointers, left and right. A doubly linked list has two pointers, previous and next. If we interpret the BST's left pointer as previous and the BST's right pointer as next, a BST's node can be used as a node in a doubly linked list. Also, the inorder traversal of a BST represents an ordered set just like a doubly linked list. Hence it is possible to take a BST and rewrite its node pointers so that it represents a doubly linked list such that the resulting list represents inorder traversal sequence of the tree.



(a) A BST of five nodes—edges that do not terminate in nodes denote empty subtrees. The number in hex adjacent to each node represents its address in memory.

(b) The sorted doubly linked list corresponding to the BST in (a). Note how the tree nodes have been used for the list nodes.

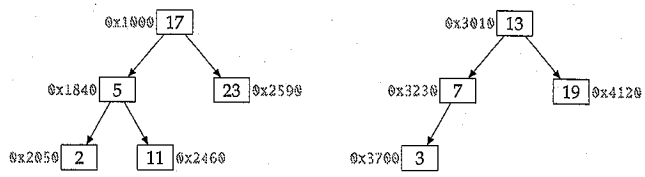
Figure 14.4: BST to sorted doubly linked list.

Problem 14.9: Design an algorithm that takes as input a BST B and returns a sorted doubly linked list on the same elements. Your algorithm should not allocate any new nodes. The original BST does not have to be preserved; use its nodes as the nodes of the resulting list, as shown in Figure 14.4 on the preceding page. *pg. 315*

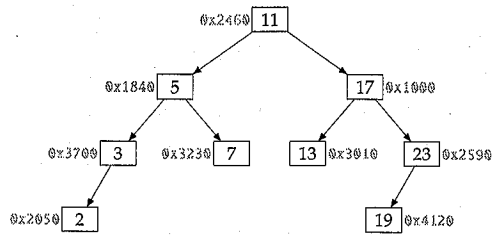
14.10 Merge two BSTs

If A and B are BSTs, it is straightforward to create a BST containing the union of their keys: traverse one, and insert its keys into the other. Many other constructions are possible; see Figure 14.5 for an example.

If both BSTs are balanced and the insertion preserves balance, the time complexity is $O(n \log n)$, where n is the total number of nodes in A and B.



(a) Two BSTs.



(b) Merged BST corresponding to the two BSTs in (a).

Figure 14.5: Example of merging two BSTs.

Problem 14.10: Let A and B be BSTs. Design an algorithm that merges them in $O(n)$ time. You cannot use dynamic allocation. You do not need to preserve the original trees. You can update pointer fields, but cannot change the key stored in a node. *pg. 316*

14.11 Find the k largest elements in a BST

A BST is a sorted data structure, which suggests that it should be possible to find the k largest keys easily.

Problem 14.11: Given the root of a BST and an integer k, design a function that finds the k largest elements in this BST. For example, if the input to your function is the BST in Figure 14.1 on Page 105 and $k = 3$, your function should return $\langle 53, 47, 43 \rangle$. *pg. 317*

14.12 Traversal orders in a BST

As discussed in Problem 9.7 on Page 77 there are many different binary trees that yield the same sequence of visited nodes in an inorder order traversal; the same is true for a preorder traversal, and a postorder traversal. For a binary tree, given an inorder traversal and any of the other two traversal orders, there exists a unique binary tree that yields those orders. However, if a binary tree satisfies the BST property, the added constraints make it possible to reconstruct the tree with less traversal information.

Problem 14.12: Which traversal orders—inorder, preorder, and postorder—of a BST can be used to reconstruct the BST uniquely? Write a program that takes as input a sequence of node keys and computes the corresponding BST. Assume that all keys are unique. *pg. 318*

14.13 Lowest common ancestor in a BST

Since a BST is a specialized binary tree, the notion of lowest common ancestor, as expressed in Problem 9.12 on Page 78, holds for BST nodes too.

In general, computing the LCA of two nodes in a BST is no easier than computing the LCA in a binary tree, since any binary tree can be viewed as a BST where all the keys are equal. However, when the keys are distinct, it is possible to improve on the LCA algorithms for binary trees.

Problem 14.13: Design an algorithm that takes a BST T of size n and height h, nodes s and b, and returns the LCA of s and b. Assume $s.key < b.key$. For example, in Figure 14.1 on Page 105, if s is node C and b is node G, your algorithm should return node B. Your algorithm should run in $O(h)$ time and $O(1)$ space. Nodes do not have pointers to their parents. *pg. 320*

14.14 Descendant and ancestor

Problem 14.14: Let r, s, and m be distinct nodes in a BST. In this BST, nodes do not have pointers to their parents and all keys are unique. Write a function which returns `true` if m has both an ancestor and a descendant in the set $\{r, s\}$. For example, in Figure 14.1 on Page 105, if m is Node J, your function should return `true` if the given set is $\{A, K\}$ and return `false` if the given set is $\{I, P\}$. *pg. 321*

14.15 Nearest restaurant

Consider the problem of developing a web-service that takes a geographical location, and returns the nearest restaurant. The service starts with a set S of n restaurant locations—each location is a pair of x, y-coordinates. A query consists of a location, and should return the nearest restaurant (ties may be broken arbitrarily).

One approach is to build two BSTs on the restaurant locations: T_X sorted on the x coordinates, and T_Y sorted on the y coordinates. A query on location (p, q) can be performed by finding all the points P_Δ whose x coordinate is in the range $[p-\Delta, p+\Delta]$,

and all the points Q_Δ whose y coordinate is in the range $[q - \Delta, q + \Delta]$, computing $R_\Delta = P_\Delta \cap Q_\Delta$ and finding the point in R_Δ closest to (p, q). Heuristically, if Δ is chosen correctly, R_Δ is a small subset of S, and a brute-force search for the closest point in R_Δ is fast. Of course, Δ has to be chosen correctly—one approach is to start with a small value and keep doubling it until R_Δ is nonempty.

This approach performs poorly on pathological data, but works well in practice. Theoretically better approaches exist, e.g., Quadtrees, which decompose the plane into regions which are balanced with respect to the number of points they contain, or k-d trees, which organize points in a k-dimensional space and provide range searches and nearest neighbor searches efficiently.

Problem 14.15: How would you efficiently perform a range query on a BST? Specifically, write a function that takes as input a BST and a range $[L, U]$ and returns a list of all the keys that lie in $[L, U]$? *pg. 322*

14.16 Minimize the distance in three sorted arrays

Let A, B, and C be sorted arrays of integers. Define distance$(i, j, k) = \max(|A[i] - B[j]|, |A[i] - C[k]|, |B[j] - C[k]|)$.

Problem 14.16: Design an algorithm that takes three sorted arrays A, B, and C and returns a triple (i, j, k) such that distance(i, j, k) is minimum. Your algorithm should run in $O(|A| + |B| + |C|)$ time. *pg. 323*

14.17 Most visited pages

You are given a log file containing billions of entries. Each entry contains an integer `timestamp` and `page` which is of type string. The entries in a log file appear in increasing order of timestamp.

Problem 14.17: You are to implement methods to analyze log file data to find the most visited pages. Specifically, implement the following methods:

- `void add(Entry p)`—add `p.page` to the set of visited pages. It is guaranteed that if `add(q)` is called after `add(p)` then `q.timestamp` is greater than or equal to `p.timestamp`.
- `List<String> common(k)`—return a list of the `k` most common pages.

First solve this problem when `common(k)` is called exactly once after all pages have been read. Then solve the problem when calls to `common` and `add` are interleaved. Assume you have unlimited RAM. *pg. 325*

14.18 Most visited pages in a window

This problem is a continuation of Problem 14.17. The difference is that only pages whose timestamps are within a specified duration of the page most recently read are to be considered.

Problem 14.18: Implement the API in Problem 14.17. If `common` is called after processing the i-th entry, `common` should return the k most visited pages whose

timestamp is in $[t_i - W, t_i]$. Here t_i is the timestamp of the i-th entry and W is specified by the client before any pages are read and does not change. RAM is limited—in particular you cannot keep a map containing all pages. Maximize time efficiency assuming calls to add and common may be interleaved and common is frequently called.

pg. 325

14.19 Gaussian primes (☺)

The Gaussian integers are complex numbers of the form $a + bi$, where a and b are integers and $i = \sqrt{-1}$. The numbers $1, -1, i$, and $-i$ are known as *units*. A nonzero Gaussian integer α is called a Gaussian prime if $\alpha = \beta\gamma \Rightarrow \beta$ is a unit or γ is a unit. Examples are given in Figure 14.6.

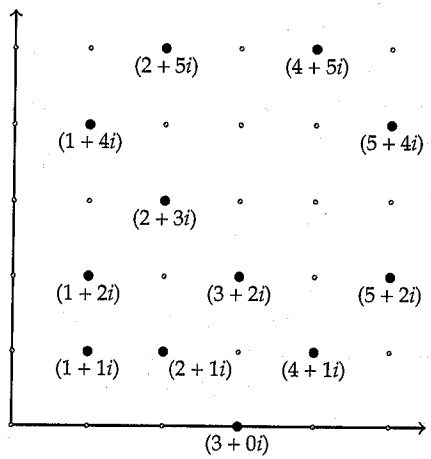


Figure 14.6: Gaussian primes with real part in $[1, 5]$ and imaginary part in $[0, 5]$. Observe $2 + 0i = (1 + i)(1 - i)$, so $2 + 0i$ is not a Gaussian prime.

Problem 14.19: Write a function that takes a single integer argument n and computes all the Gaussian integers $a + bi$, for $-n \le a, b \le n$ that are Gaussian primes. *pg. 326*

14.20 View from above (☺)

This is a simplified version of a problem that often comes up in computer graphics.

You are given a set of line segments. Each segment consists of a closed interval $[l_i, r_i]$ of the x-axis, a color, and a height. When viewed from above, the color at point x on the x-axis is the color of the highest segment that includes x. This is illustrated in Figure 14.7 on the following page.

Problem 14.20: Implement a function that computes the view from above. Your input is a sequence of line segments, each specified as a 4-tuple $\langle l, r, c, h \rangle$, where l and r are the left and right endpoints, respectively, c encodes the color, and h are the height. The output should be in the same format. No two segments whose intervals overlap have the same height. *pg. 327*

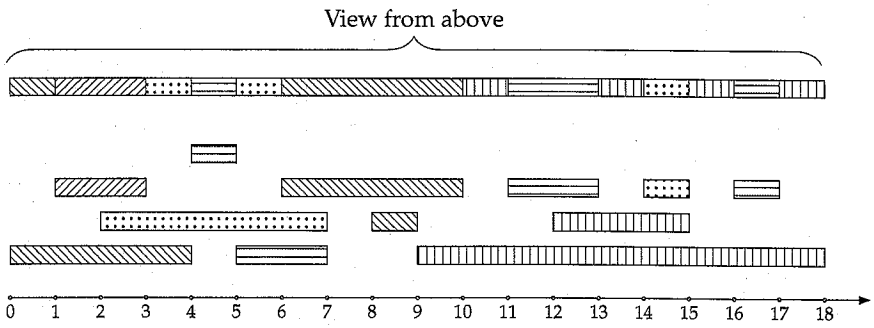


Figure 14.7: Instance of the view from above problem, with patterns used to denote colors.

Augmented BSTs

Thus far we have considered BSTs in which each node stores an entry drawn from a sorted set, a left child, a right child, and a parent. Adding fields to the nodes can speed up certain queries, as the following problems illustrate.

14.21 Adding credits

Consider a server that a large number of clients connect to. Each client is identified by a unique string. Each client has a certain number of "credits", which is a nonnegative integer value. The server needs to maintain a data structure to which clients can be added, removed, queried, or updated. In addition, the server needs to be able to add C credits to all clients simultaneously.

Problem 14.21: Design a data structure that implements the following methods:

- `insert(s,c)`, which adds client s with credit c, overwriting any existing entry for s.
- `remove(s)`, which removes client s.
- `lookup(s)`, which returns the number of credits associated with client s, or -1 if s is not present.
- `addAll(C)`, the effect of which is to increment the number of credits for each client currently present by C.
- `max()`, which returns the client with the highest number of credits.

The `insert(s,c)`, `remove(s)`, and `lookup(s)` methods should run in time $O(\log n)$, where n is the number of clients. The remaining methods should run in time $O(1)$.

pg. 330

14.22 Counting the number of entries in an interval

One problem with the approach to the restaurant problem outlined in Problem 14.15 on Page 109 is that the number of entries in P_Δ could be much larger than the number of entries in Q_Δ or vice versa. We can address this by first computing the number of entries that lie in a range.

Problem 14.22: Suppose each node in a BST has a size field, which denotes the number of nodes at the subtree rooted at that node, inclusive of the node. How would you efficiently compute the number of nodes that lie in a given range? Can the size field be updated efficiently on insert and on delete? *pg. 331*

14.23 Querying server logs

Consider the problem of analyzing web server logs. The logs contain information on sessions by individual users, including the time when their session began, and when it ended.

Problem 14.23: Design a data structure that stores closed intervals and can efficiently return the complete set of intervals that intersect a specified range $[L, U]$. Your data structure must also support efficient insertions and deletions. *pg. 332*

CHAPTER

15

Meta-algorithms

The important fact to observe is that we have attempted to solve a maximization problem involving a particular value of x and a particular value of N by first solving the general problem involving an arbitrary value of x and an arbitrary value of N.

— *"Dynamic Programming,"*
R. E. BELLMAN, 1957

We now cover three general techniques for algorithm design—*divide and conquer*, *dynamic programming*, and the *greedy method*. The approaches described previously, such as mapping a problem into an appropriate data structure, or presorting the input, are more widely used than the methods in this chapter. However, although they are specialized, the approaches in this chapter lead to huge efficiency gains compared to naïve algorithms. These techniques are not exhaustive. In later chapters we will discuss algorithms that use randomization, parallelization, backtracking, heuristic search, reduction, and approximation.

Divide and conquer

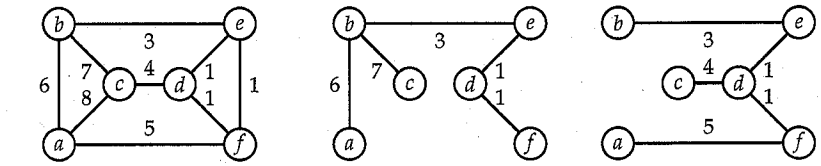
A divide and conquer algorithm works by repeatedly decomposing a problem into two or more smaller independent subproblems of the same kind, until it gets to instances that are simple enough to be solved directly. The solutions to the subproblems are then combined to give a solution to the original problem.

Merge sort and quicksort are classical examples of divide and conquer. In merge sort, the array $A[0 : n-1]$ is sorted by sorting $A[0 : \lfloor n/2 \rfloor]$ and $A[\lfloor n/2 \rfloor + 1 : n-1]$, and merging them. In quicksort, $A[0 : n-1]$ is sorted by selecting a pivot element $A[r]$ and reordering the elements of A to make all elements appearing before $A[r]$ less than or equal to $A[r]$ and all elements appearing after $A[r]$ greater than or equal to $A[r]$. The subarray consisting of elements before $A[r]$ and the subarray consisting of elements after $A[r]$ are sorted, and the resulting array is completely sorted.

Interestingly, the divide step in merge sort is trivial; the challenge is in combining the results. With quicksort, the opposite is true. Problems 10.1 on Page 80 and 6.1 on Page 52 illustrate the key computations in merge sort and quicksort.

A divide and conquer algorithm is not always optimum. A minimum spanning tree (MST) is a minimum weight set of edges in a weighted undirected graph which connect all vertices in the graph; refer to Problems 16.13 on Page 137 and 17.6 on Page 141 for details. A natural divide and conquer algorithm for computing the MST

is to partition the vertex set V into two subsets V_1 and V_2, compute MSTs for V_1 and V_2, and then join these two MSTs with an edge of minimum weight between V_1 and V_2. Figure 15.1 shows how this algorithm can lead to suboptimal results.



(a) A weighted undirected graph. (b) An MST built from the MSTs on $\{a, b, c\}$ and $\{d, e, f\}$. (c) An optimum MST.

Figure 15.1: Divide and conquer applied to the MST problem is suboptimum.

The term divide and conquer is also sometimes applied to algorithms that reduce a problem to only one subproblem, e.g., binary search. Such algorithms can be implemented more efficiently than general divide and conquer algorithms. In particular, these algorithms use tail recursion, which can be replaced by a loop. Decrease and conquer is a more appropriate term for such algorithms.

15.1 Drawing the skyline (🐞)

A number of buildings are visible from a point. Each building is a rectangle, and the bottom of each building lies on a fixed line. A building is specified using its left and right coordinates, and its height. One building may partly obstruct another, as shown in Figure 15.2(a). The skyline is the list of coordinates and corresponding heights of what is visible.

For example, the skyline corresponding to the buildings in Figure 15.2(a) is given in Figure 15.2(b). (The patterned rectangles illustrate the largest rectangle under the skyline problem, which is described in Problem 15.8 on Page 120, and is not relevant to the current problem.)

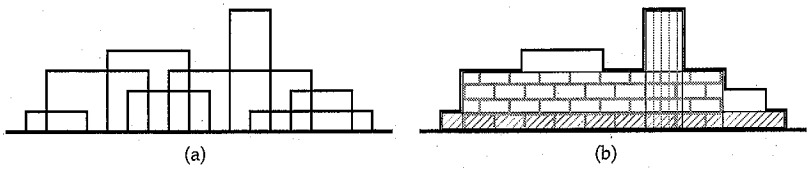


Figure 15.2: Buildings, their skyline, and the largest contained rectangle.

Problem 15.1: Design an efficient algorithm for computing the skyline. *pg. 332*

15.2 Counting inversions (🐞)

Let A be an array of n numbers. The pair of indices (i, j) is said to be inverted if $i < j$ and $A[i] > A[j]$.

Problem 15.2: Design an efficient algorithm that takes an array A of n numbers and returns the number of inverted pairs of indices. *pg. 334*

15.3 Nearest points in the plane

Suppose you were asked to design a collision warning system for a ship control system. Specifically, your program receives coordinates for the different ships, and has to compute the pair of ships that is at greatest risk of collision. Assuming that the pair with the greatest risk is the pair that is closest, your problem then becomes the following.

Problem 15.3: You are given a list of pairs of points in the two-dimensional Cartesian plane. Each point has integer x and y coordinates. How would you find the two closest points? *pg. 335*

15.4 Tree diameter

Packets in Ethernet local area networks (LANs) are routed according to the unique path in a tree whose nodes correspond to clients and edges correspond to physical connections between the clients. In this problem, we want to design an algorithm for finding the "worst-case" route, i.e., the two clients that are furthest apart. In the abstract, we want to solve the following problem:

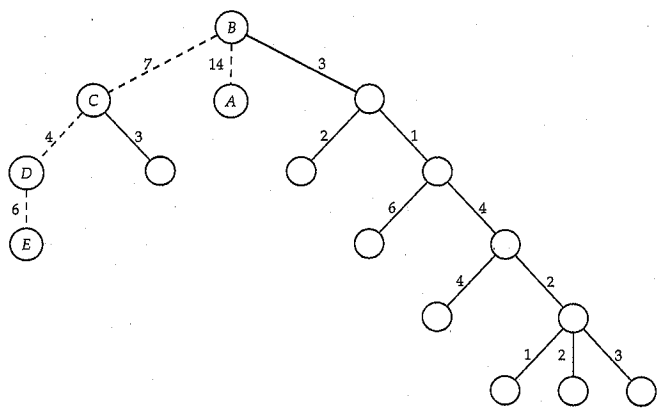


Figure 15.3: The diameter for the above tree is 31. The corresponding path is $\langle A, B, C, D, E \rangle$, which is depicted by the dashed edges.

Let T be a tree, where each edge is labeled with a nonnegative real-valued distance. Define the diameter of T to be the length of a longest path in T. Figure 15.3 illustrates the diameter concept.

Problem 15.4: Design an efficient algorithm to compute the diameter of a tree. *pg. 337*

Dynamic programming

DP is a general technique for solving complex optimization problems that can be decomposed into overlapping subproblems. Like divide and conquer, we solve the problem by combining the solutions of multiple smaller problems but what makes DP different is that the subproblems may not be independent. A key to making DP efficient is reusing the results of intermediate computations. (The word "programming" in dynamic programming does not refer to computer programming—the word was chosen by Richard Bellman to describe a program in the sense of a schedule.) Problems which are naturally solved using DP are a popular choice for hard interview questions.

To illustrate the idea underlying DP, consider the problem of computing Fibonacci numbers defined by $F_n = F_{n-1} + F_{n-2}$, $F_0 = 0$ and $F_1 = 1$. A function to compute F_n that recursively invokes itself to compute F_{n-1} and F_{n-2} would have a time complexity that is exponential in n. However if we make the observation that recursion leads to computing F_i for $i \in [0, n-1]$ repeatedly, we can save the computation time by storing these results and reusing them. This makes the time complexity linear in n, albeit at the expense of $O(n)$ storage. Note that the recursive implementation requires $O(n)$ storage too, though on the stack rather than the heap and that the function is not tail recursive since the last operation performed is + and not a recursive call.

The key to solving any DP problem efficiently is finding the right way to break the problem into subproblems such that

- the bigger problem can be solved relatively easily once solutions to all the subproblems are available, and
- you need to solve as few subproblems as possible.

In some cases, this may require solving a slightly different optimization problem than the original problem. For example, consider the following problem: given an array of integers A of length n, find the interval indices a and b such that $\sum_{i=a}^{b} A[i]$ is maximized. As a concrete example, the interval corresponding to the maximum subarray sum for the array in Figure 15.4 is $[0, 3]$.

904	40	523	12	-335	-385	-124	481	-31
$A[0]$	$A[1]$	$A[2]$	$A[3]$	$A[4]$	$A[5]$	$A[6]$	$A[7]$	$A[8]$

Figure 15.4: An array with a maximum subarray sum of 1479.

The brute-force algorithm, which computes each subarray sum, has $O(n^3)$ time complexity—there are $\frac{n(n-1)}{2}$ subarrays, and each subarray sum can be computed in $O(n)$ time. The brute-force algorithm can be improved to $O(n^2)$ by first computing sums $S[i]$ for subarrays $A[0 : i]$ for each $i < n$; the sum of subarray $A[i : j]$ is $S[j] - S[i-1]$, where $S[-1]$ is taken to be 0.

Here is a natural divide and conquer algorithm. We solve the problem for the subarrays $L = A[0 : \lfloor \frac{n}{2} \rfloor]$ and $R = A[\lfloor \frac{n}{2} \rfloor + 1 : n-1]$. In addition to the answers for each, we also return the maximum subarray sum for any subarray ending at $|L|-1$ for

L (call this value l) and starting at 0 for R (call this value r). The maximum subarray sum for A is the maximum of $l + r$, the answer for L, and the answer for R. The time complexity analysis is similar to that for quicksort, which leads to a $O(n \log n)$.

Now we will solve this problem by using DP. A natural thought is to assume we have the solution for the subarray $A[0 : n-2]$. However, even if we knew the largest sum subarray for subarray $A[0 : n-2]$, it does not help us solve the problem for $A[0 : n-1]$. A better approach is to iterate through the array. For each index j, the maximum subarray ending at j is equal to $S[j] - \min_{i \le j} S[i]$. During the iteration, we cache the minimum subarray sum we have visited and compute the maximum subarray for each index. The time spent per index is constant, leading to an $\Theta(n)$ time and $O(1)$ space solution. Following is the code in C++:

```
template <typename T>
pair<int, int> find_maximum_subarray(const vector<T> &A) {
  // A[range.first : range.second - 1] will be the maximum subarray
  pair<int, int> range(0, 0);
  int min_idx = -1;
  T min_sum = 0, sum = 0, max_sum = numeric_limits<T>::min();
  for (int i = 0; i < A.size(); ++i) {
    sum += A[i];
    if (sum < min_sum) {
      min_sum = sum, min_idx = i;
    }
    if (sum - min_sum > max_sum) {
      max_sum = sum - min_sum, range = {min_idx + 1, i + 1};
    }
  }
  return range;
}
```

Here are two variants of the subarray maximization problem that can be solved with ideas that are similar to the above approach: find indices a and b such that $\sum_{i=a}^{b} A[i]$ is—(1.) closest to 0 and (2.) closest to t. (Both entail some sorting, which increases the time complexity to $O(n \log n)$.) Another good variant is finding indices a and b such that $\prod_{i=a}^{b} A[i]$ is maximum when the array contains both positive and negative integers.

A common mistake in solving DP problems is trying to think of the recursive case by splitting the problem into two equal halves, *a la* quicksort, i.e., somehow solve the subproblems for subarrays $A[0 : \lfloor \frac{n}{2} \rfloor]$ and $A[\lfloor \frac{n}{2} \rfloor + 1 : n]$ and combine the results.

A common mistake in solving DP problems is trying to think of the recursive case by splitting the problem into two equal halves, *a la* quicksort, i.e., somehow solve the subproblems for subarrays $A[0 : \lfloor \frac{n}{2} \rfloor]$ and $A[\lfloor \frac{n}{2} \rfloor + 1 : n]$ and combine the results. However in most cases, these two subproblems are not sufficient to solve the original problem.

15.5 Maximum subarray sum in a circular array

Finding the maximum subarray sum in an array can be solved in linear time, as described on the previous page. However, if the given array A is circular, which

means the first and last elements of the array are to be treated as being adjacent to each other, the algorithm yields suboptimum solutions. For example, if A is the array in Figure 15.4 on Page 117, the maximum subarray sum starts at index 7 and ends at index 3, but the algorithm described on the preceding page returns the subarray from index 0 to index 3.

Problem 15.5: Given a circular array A, compute its maximum subarray sum in $O(n)$ time, where n is the length of A. Can you devise an algorithm that takes $O(n)$ time and $O(1)$ space? *pg. 339*

15.6 Longest nondecreasing subsequence (☠)

The problem of finding the longest nondecreasing subsequence in a sequence of integers has implications to many disciplines, including string matching and analyzing card games. As a concrete instance, the length of a longest nondecreasing subsequence for the array A in Figure 15.5 is 4. There are multiple longest nondecreasing subsequences, e.g., $\langle 0, 4, 10, 14 \rangle$ and $\langle 0, 2, 6, 9 \rangle$.

0	8	4	12	2	10	6	14	1	9
$A[0]$	$A[1]$	$A[2]$	$A[3]$	$A[4]$	$A[5]$	$A[6]$	$A[7]$	$A[8]$	$A[9]$

Figure 15.5: An array whose longest nondecreasing subsequences are of length 4.

Problem 15.6: Given an array A of n numbers, find a longest subsequence $\langle i_0, \ldots, i_{k-1} \rangle$ such that $i_j < i_{j+1}$ and $A[i_j] \leq A[i_{j+1}]$ for any $j \in [0, k-2]$. *pg. 340*

15.7 Longest subarray whose sum $\leq k$ (☠)

Here we consider finding the longest subarray subject to a constraint on the subarray sum. For example, for the array in Figure 15.6, the longest subarray whose subarray sum is no more than 184 is $A[3:6]$.

431	-15	639	342	-14	565	-924	635	167	-70
$A[0]$	$A[1]$	$A[2]$	$A[3]$	$A[4]$	$A[5]$	$A[6]$	$A[7]$	$A[8]$	$A[9]$

Figure 15.6: An array for the longest subarray whose sum $\leq k$ problem.

Problem 15.7: Design an algorithm that takes as input an array A of n numbers and a key k, and returns a longest subarray of A for which the subarray sum is less than or equal to k. *pg. 342*

15.8 Largest rectangle under the skyline

You are given a sequence of adjacent buildings. Each has unit width and an integer height. These buildings form the skyline of a city. An architect wants to know the area of a largest rectangle contained in this skyline. For example, for the skyline in Figure 15.2(b) on Page 115, the largest rectangle is the brick-patterned one. Note that it is not the contained rectangle with maximum height (which is denoted by the vertical-patterning), or the maximum width (which is denoted by the slant-patterning).

Problem 15.8: Let A be an array of n numbers encoding the heights of adjacent buildings of unit width. Design an algorithm to compute the area of the largest rectangle contained in this skyline, i.e., compute $\max_{i<j}((j-i+1) \times \min_{k=i}^{j} A[k])$.

pg. 344

15.9 Maximum 2D subarray

The following problem has applications to image processing.

Problem 15.9: Let A be an $n \times m$ Boolean 2D array. Design efficient algorithms for the following two problems:

- What is the largest 2D subarray containing only 1s?
- What is the largest square 2D subarray containing only 1s?

What are the time and space complexities of your algorithms as a function of n and m?

pg. 345

15.10 Searching for a sequence in a 2D array

Let A be a 2D array of integers, and S a 1D array of integers. We say S *occurs* in A if you can start from some entry in A and traverse adjacent entries in A in the order prescribed by S till you get to the end of S. The entries adjacent to $A[i][j]$ are $A[i-1][j], A[i+1][j], A[i][j-1]$, and $A[i][j+1]$, assuming the indices are valid. It is acceptable to visit an entry in A more than once.

For example, if

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 3 & 4 & 5 \\ 5 & 6 & 7 \end{bmatrix}$$

and $S_1 = \langle 1,3,4,6 \rangle$, then S_1 occurs in A—consider the entries $\langle A[0][0], A[1][0], A[1][1], A[2][1] \rangle$. However $S_2 = \langle 1,2,3,4 \rangle$ does not occur in A.

Problem 15.10: Design an algorithm that takes as arguments a 2D array A and a 1D array S, and determines whether S appears in A. If S appears in A, print the sequence of entries where it appears.

pg. 348

15.11 Levenshtein distances

Spell checkers make suggestions for misspelled words. Given a misspelled string s, a spell checker should return words in the dictionary which are close to s.

In 1965, Vladimir Levenshtein defined the distance between two words as the minimum number of "edits" it would take to transform the misspelled word into a correct word, where a single edit is the *insertion*, *deletion*, or *substitution* of a single character.

Problem 15.11: Given two strings, represented as arrays of characters A and B, compute the minimum number of edits needed to transform the first string into the second string. *pg. 349*

15.12 Word breaking

Suppose you are designing a search engine. In addition to getting keywords from a page's content, you would like to get keywords from Uniform Resource Locators (URLs). For example, bedbathandbeyond.com should be associated with "bed bath and beyond" (in this version of the problem we also allow "bed bat hand beyond" to be associated with it).

Problem 15.12: Given a dictionary and a string s, design an efficient algorithm that checks whether s is the concatenation of a sequence of dictionary words. If such a concatenation exists, your algorithm should output it. *pg. 351*

15.13 Pretty printing

Consider the problem of laying out text using a fixed width font. Each line can hold no more than L characters. Words on a line are to be separated by exactly one blank. Therefore we may be left with white space at the end of a line (since the next word will not fit in the remaining space). This white space is visually unappealing.

Define the *messiness* of the end-of-line whitespace as follows. The messiness of a line ending with b blank characters is 2^b. The total messiness of a sequence of lines is the sum of the messinesses of all the lines. A sequence of words can be split across lines in different ways with different messiness, as illustrated in Figure 15.7.

```
I have inserted a large number of␣␣␣
new examples from the papers for the
Mathematical Tripos during the last␣
twenty years, which should be useful
to Cambridge students.␣␣␣␣␣␣␣␣␣␣␣␣␣
```

(a) Messiness = $2^3 + 2^0 + 2^1 + 2^0 + 2^{14} = 16396$.

```
I have inserted a large number␣␣␣␣␣␣
of new examples from the papers␣␣␣␣␣
for the Mathematical Tripos during␣␣
the last twenty years, which should␣
be useful to Cambridge students.␣␣␣␣
```

(b) Messiness = $2^6 + 2^5 + 2^2 + 2^1 + 2^4 = 118$.

Figure 15.7: Two layouts for the same sequence of words; the line length L is 36.

Problem 15.13: Given text, i.e., a string of words separated by single blanks, decompose the text into lines such that no word is split across lines and the messiness of the decomposition is minimized. Each line can hold no more than L characters. How would you change your algorithm if the messiness is the sum of the messinesses of all but the last line? *pg. 352*

15.14 Computing the binomial coefficients

The symbol $\binom{n}{k}$ is short form for $\frac{n(n-1)\cdots(n-k+1)}{k(k-1)\cdots(3)(2)(1)}$. It is the number of ways to choose a k-element subset from an n-element set.

It is not obvious that the expression defining $\binom{n}{k}$ always yields an integer. Furthermore, direct computation of $\binom{n}{k}$ from this expression quickly results in the numerator or denominator overflowing if integer types are used, even if the final result fits in a 32-bit integer. If floats are used, the expression may not yield a 32-bit integer.

Problem 15.14: Design an efficient algorithm for computing $\binom{n}{k}$ which has the property that it never overflows if $\binom{n}{k}$ can be represented as a 32-bit integer; assume n and k are integers. *pg. 353*

15.15 Score combinations

In an American football game, a play can lead to 2 points (safety), 3 points (field goal), or 7 points (touchdown). Given the final score of a game, we want to compute how many different combinations of 2, 3, and 7 point plays could make up this score.

For example, if $W = \{2, 3, 7\}$, four combinations of plays yield a score of 12:

- 6 safeties ($2 \times 6 = 12$),
- 3 safeties and 2 field goals ($2 \times 3 + 3 \times 2 = 12$),
- 1 safety, 1 field goal and 1 touchdown ($2 \times 1 + 3 \times 1 + 7 \times 1 = 12$), and
- 4 field goals ($3 \times 4 = 12$).

Problem 15.15: You have an aggregate score s and W which specifies the points that can be scored in an individual play. How would you find the number of combinations of plays that result in an aggregate score of s? How would you compute the number of distinct sequences of individual plays that result in a score of s? *pg. 354*

15.16 Number of ways

Suppose you start at the top-left corner of an $n \times m$ 2D array A and want to get to the bottom-right corner. The only way you can move is by either going right or going down. Three legal paths for a 5×5 2D array are given in Figure 15.8.

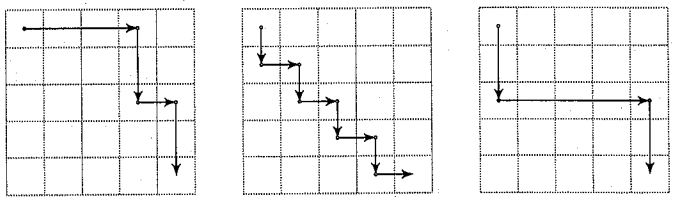


Figure 15.8: Paths through a 2D array.

Problem 15.16: How many ways can you go from the top-left to the bottom-right in an $n \times m$ 2D array? How would you count the number of ways in the presence of obstacles, specified by an $n \times m$ Boolean 2D array B, where a `true` represents an obstacle. *pg. 356*

15.17 Planning a fishing trip

A fisherman is in a rectangular sea. The value of the fish at point (i, j) in the sea is specified by an $n \times m$ 2D array A.

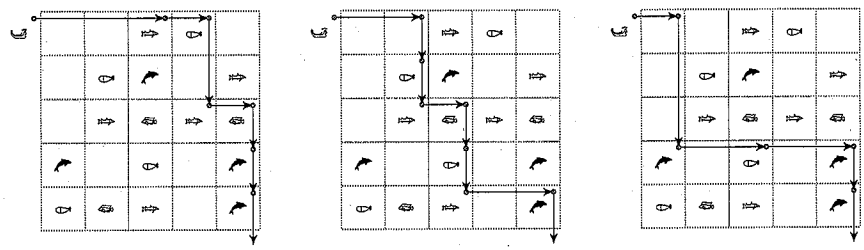


Figure 15.9: Alternate paths for a fisherman. Different types of fish have different values, which are known to the fisherman.

Problem 15.17: Write a program that computes the maximum value of fish a fisherman can catch on a path from the upper leftmost point to the lower rightmost point. The fisherman can only move down or right, as illustrated in Figure 15.9. *pg. 357*

15.18 Picking up coins, maximum gain

In the pick-up-coins game, an even number of coins are placed in a line, as in Figure 4.6 on Page 44. Two players, F and S, take turns at choosing one coin each—they can only choose from the two coins at the ends of the line. Player F goes first. The game ends when all the coins have been picked up. The player whose coins have the higher total value wins. A player cannot pass his turn.

Problem 15.18: Design an efficient algorithm for computing the maximum margin of victory for the starting player in the pick-up-coins game. *pg. 357*

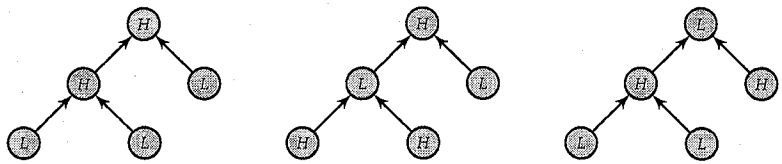
15.19 Voltage selection in a logic circuit

A logic circuit is an ensemble of logic gates operating on a set of external inputs. The gates implement basic Boolean operations such as AND, OR and NOT. Formally, a logic circuit can be modeled as a directed acyclic graph (DAG)—the external inputs are the sources of the DAG and gates are the remaining nodes.

In this problem we consider the special case where the DAG is a rooted tree. Each node can use either a high voltage or a low voltage. A low voltage node consumes less power, but has a weaker signal. It is a design constraint that a low voltage node should never be input to another low voltage node. Let c_v be the number of children. The power used by a low voltage node is $c_v + 1$; the power used by a high voltage node is $2(c_v + 1)$.

Figure 15.10 on the next page shows three voltage assignments for the same logic circuit. All assignments satisfy the design constraint. Figure 15.10(a) on the following page proceeds greedily in a bottom up fashion, assigning leaves to L. Figure 15.10(b)

proceeds greedily in a top-down fashion, assigning the root to H, its children to L, and continuing downwards. Figure 15.10(c) is the optimal assignment.



(a) Power = 1+2×3+1+2×3+1 = 15 (b) Power = $2 \times 1 + 3 + 2 \times 1 + 2 \times 3 + 1 = 14$ (c) Power = 1+2×3+1+3+2×1 = 13

Figure 15.10: The node labels L or H indicating a high or low voltage node, respectively.

Problem 15.19: Design an algorithm for minimizing power that takes as input a rooted tree and assigns each node to a low or high voltage, subject to the design constraint. *pg. 359*

15.20 Image compression

Suppose a rectangular black-and-white image is formally represented by an $m \times n$ Boolean 2D array P. Intuitively, if large contiguous regions of the image all have the same color, the array representation is suboptimum from the perspective of memory usage.

A two-dimensional tree is a data structure that can be used to represent a partition of a rectangle into subrectangles. Formally, a two-dimensional tree is either a monochromatic rectangle, or consists of a *root* node r, the lower leftmost and upper rightmost points (a, b) and (c, d) of the corresponding rectangle, a splitting point (s, t), and an ordered list of four two-dimensional trees SW, NW, NE, and SE representing 2D subarrays $P[a : s-1, b : t-1]$, $P[a : s-1, t : d]$, $P[s : c, t : d]$, and $P[s : c, b : t-1]$, respectively.

Problem 15.20: Implement cutpoint selection to minimize the number of nodes in the two-dimensional tree representing an image. *pg. 359*

The greedy method

As described on Page 32, the greedy method is an algorithm design pattern which results in an algorithm that computes a solution in steps. At each step the algorithm makes a decision that is locally optimum, and never changes that decision.

The example on Page 32 illustrates how different greedy algorithms for the same problem can differ in terms of optimality. As another example, consider making change for 48 pence in the old British currency where the coins came in $30, 24, 12, 6, 3$, and 1 pence denominations. Suppose our goal is to make change using the smallest number of coins. The natural greedy algorithm iteratively chooses the largest denomination coin that is less than or equal to the amount of change that

remains to be made. If we try this for 48 pence, we get three coins—30 + 12 + 6. However the optimum answer would be two coins—24 + 24.

In its most general form, the coin changing problem is NP-hard (Chapter 17) but for some coinages, the greedy algorithm is optimum—e.g., if the denominations are of the form $\{1, r, r^2, r^3\}$. (An *ad hoc* argument can be applied to show that the greedy algorithm is also optimum for US coinage.) The general problem can be solved in pseudo-polynomial time using DP in a manner similar to Problem 17.2 on Page 139.

As another example of how greedy reasoning can fail, consider the following problem: Four travelers need to cross a river as quickly as possible in a small boat. Only two people can cross at one time. The speed to cross the river is dictated by the slower person in the boat (if there is just one person, that is his speed). The four travelers have times of 5, 10, 20, and 25 minutes. The greedy schedule would entail having the two fastest travelers cross initially (10), with the fastest returning (5), picking up the faster of the two remaining and crossing again (20), and with the fastest returning for the slowest traveler (5 + 25). The total time taken would be 10 + 5 + 20 + 5 + 25 = 65 minutes. However, a better approach would be for the fastest two to cross (10), with the faster traveler returning (5), and then having the two slowest travelers cross (25), with the second fastest returning (10) to pick up the fastest traveler (10). The total time for this schedule is 10 + 5 + 25 + 10 + 10 = 60 minutes.

15.21 Minimize waiting time (☺)

A database has to respond to n simultaneous client SQL queries. The service time required for Query i, where $1 \leq i \leq n$, equals t_i milliseconds and is known in advance. The query lookups are processed by the database one at a time, but can be done in any order. It is natural to minimize the total waiting time $\sum_{i=1}^{n} T_i$, where T_i is the time at which processing for Query i begins. For example, if the lookups are done in order of increasing i, then the waiting time for the i-th query is $T_i = \sum_{j=1}^{i-1} t_j$ milliseconds.

Problem 15.21: Given n queries, compute an order in which to process queries that minimizes the total waiting time. *pg. 362*

15.22 Scheduling tutors

You are the coordinator of a tutoring service. Each day you receive requests for lessons. Each lesson has a specified start time between 9:00 a.m. and 5:00 p.m. and lasts exactly 30 minutes. You have access to an unlimited number of tutors. Tutors can start work at any time, but must stop tutoring for the day at most two hours after starting. A tutor can conduct only one lesson at a time.

Problem 15.22: Design an algorithm that computes the least number of tutors needed to schedule a set of requests. *pg. 362*

15.23 Job assignment

We have m tasks and n servers. Task i consists of $T[i]$ jobs, where a job is a unit of work. There are no dependencies between jobs in a task. Server j can execute $S[j]$ units of work in a unit time. Each $T[i]$ and $S[j]$ is a positive integer. We want to find an assignment of jobs to servers subject to the constraint that no server should receive more than one job from a single task.

Problem 15.23: Design an algorithm that takes as input a pair of arrays specifying jobs per task and server capacities, and returns an assignment of jobs to servers for which all tasks complete within one unit time. No server may process more than one job for a given task. If no such assignment exists, your algorithm should indicate that. *pg. 363*

15.24 Load balancing

Suppose you want to build a large distributed storage system on the web. Millions of users will store terabytes of data on your servers. One way to design the system would be to compute a hash code for each user's login ID, partition the hash codes across equal-sized buckets, and store the data for each bucket of users on one server. For this scheme, mapping a user to his server entails evaluating a hash function.

However if a small number of users occupy a large fraction of the storage space, this scheme will not achieve a balanced partition. One way to solve this problem is to use a nonuniform partitioning.

Problem 15.24: You have n users with unique hash codes h_0 through h_{n-1}, and m servers. The hash codes are ordered by index, i.e., $h_i < h_{i+1}$ for $i \in [0, n-2]$. User i requires b_i bytes of storage. The values $k_0 < k_1 < \cdots < k_{m-2}$ are used to assign users to servers. Specifically, the user with hash code c gets assigned to the server with the lowest ID i such that $c \leq k_i$, or to server $m-1$ if no such i exists. The load on a server is the sum of the bytes of storage of all users assigned to that server. Compute values for $k_0, k_1, \ldots, k_{m-1}$ that minimizes the load on the most heavily loaded server. *pg. 365*

15.25 Packing for USPS priority mail

The United States Postal Services (USPS) makes fixed-size mail shipping boxes—you pay a fixed price for a given box and can ship anything you want that fits in the box. Suppose you have a set of n items that you need to ship and have a large supply of the $4 \times 12 \times 8$ inch priority mail shipping boxes. Each item will fit in such a box but all of them combined will take multiple boxes. Naturally, you want to minimize the number of boxes you use.

The first-fit heuristic is a greedy algorithm for the packing problem—it maintains a sequence of boxes, and processes items to pack in the sequence in which they are given. Items are placed in the first box in which they fit.

Problem 15.25: Implement first-fit to run in $O(n \log n)$ time. *pg. 366*

15.26 Huffman coding (☠)

One way to compress a large text is by building a code book which maps each character to a bit string, referred to as its code word. Compression consists of concatenating the bit strings for each character to form a bit string for the entire text.

When decompressing the string, we read bits until we find a string that is in the code book and then repeat this process until the entire text is decoded. For the compression to be reversible, it is sufficient that the code words have the property that no code word is a prefix of another. For example, 011 is a prefix of 0110 but not a prefix of 1100.

Since our objective is to compress the text, we would like to assign the shorter strings to more common characters and the longer strings to less common characters. We will restrict our attention to individual characters. (We may achieve better compression if we examine common sequences of characters, but this increases the time complexity.)

The intuitive notion of commonness is formalized by the *frequency* of a character which is a number between zero and one. The sum of the frequencies of all the characters is 1. The average code length is defined to be the sum of the product of the length of each character's code word with that character's frequency. Table 15.1 shows the large variation in the frequencies of letters of the English alphabet.

Table 15.1: English characters and their frequencies, expressed as percentages, in everyday documents.

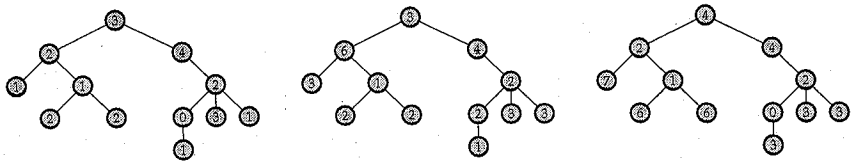
Character	Frequency	Character	Frequency	Character	Frequency
a	8.17	j	0.15	s	6.33
b	1.49	k	0.77	t	9.06
c	2.78	l	4.03	u	2.76
d	4.25	m	2.41	v	0.98
e	12.70	n	6.75	w	2.36
f	2.23	o	7.51	x	0.15
g	2.02	p	1.93	y	1.97
h	6.09	q	0.10	z	0.07
i	6.97	r	5.99		

Problem 15.26: Given a set of symbols with corresponding frequencies, find a code book that has the smallest average code length. *pg. 368*

15.27 Node reweighing (☠)

Let T be a rooted tree with n nodes. Each node u has a nonnegative weight $w(u)$. The weight of a path is the sum of the weights of the nodes on the path.

Problem 15.27: How would you efficiently assign to each node u a new weight $w'(u)$ such that (1.) each root-to-leaf path has the same weight W^*, (2.) for all nodes u, $w'(u) \geq w(u)$, and (3.) $\sum_{u \in \text{nodes}(T)} w'(u)$ is minimum? See Figure 15.11 on the next page for an example. *pg. 372*



(a) Initial weights. (b) Optimum weights, cost = 32. (c) Suboptimum weights, cost = 41.

Figure 15.11: An instance of the node reweighing problem, and two reweighings that satisfy the problem constraints. Each node is labeled with its weight.

15.28 Planning a party

Leona is holding a party and is trying to select people to invite from her circle of friends. She has n friends and knows for each pair of her friends if the first friend already knows the second friend. Leona wants to invite as many friends as possible, subject to the constraint that each invitee knows at least three other invitees and does not know at least three other invitees. For example, each of the eight men in Figure 15.12 knows three other men and does not know four other men, and this is the largest set that can be invited.

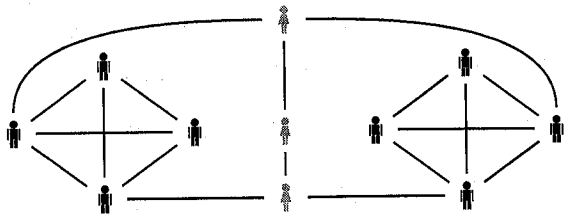


Figure 15.12: A set of eleven people. An edge between two people indicates they know each other. (For this example, A knows B iff B knows A.)

Problem 15.28: Devise an efficient algorithm that takes as input a set P of people and a set $F \subset P \times P$ of pairs of people and returns a largest subset of P within which each individual knows three or more other members of P and does not know three or more other members of P. The "knows" relation is not necessarily symmetric or transitive. *pg. 372*

15.29 Assigning radio frequencies

If two neighboring radio stations are transmitting at the same radio frequency, there would be a region geographically between them where the signal from both stations would be equally strong and the resulting interference would cause neither of the signals to be usable. Hence neighboring radio stations try to pick different frequencies. Consider the problem where we have just two frequencies available and we are given the neighborhood graph of a set of radio stations. Ideally, we want to assign

the frequencies to the radio stations such that the interference is minimized. This is a difficult problem. Suppose we are interested in a simpler problem where all we want is that for any given radio station, the majority of its neighbors use a different frequency from the given station. This can be modeled as a graph coloring problem.

Problem 15.29: Let $G = (V, E)$ be an undirected graph. A two-coloring of G is a function assigning each vertex of G to *black* or *white*. Call a two-coloring *diverse* if each vertex has at least half its neighbors opposite in color to itself. Does every graph have a diverse coloring? How would you compute a diverse coloring, if it exists?

pg. 373

CHAPTER

16 Algorithms on Graphs

Concerning these bridges, it was asked whether anyone could arrange a route in such a way that he would cross each bridge once and only once.

— *"The solution of a problem relating to the geometry of position,"*
L. EULER, 1741

Informally, a graph is a set of vertices and connected by edges. Formally, a directed graph is a tuple (V, E), where V is a set of *vertices* and $E \subset V \times V$ is the set of edges. Given an edge $e = (u, v)$, the vertex u is its *source*, and v is its *sink*. Graphs are often decorated, e.g., by adding lengths to edges, weights to vertices, and a start vertex. A directed graph can be depicted pictorially as in Figure 16.1.

A *path* in a directed graph from u to vertex v is a sequence of vertices $\langle v_0, v_1, \ldots, v_{n-1} \rangle$ where $v_0 = u$, $v_{n-1} = v$, and $(v_i, v_{i+1}) \in E$ for $i \in \{0, \ldots, n-2\}$. The sequence may contain of a single vertex. The *length* of the path $\langle v_0, v_1, \ldots, v_{n-1} \rangle$ is $n - 1$. Intuitively, the *length* of a path is the number of edges it traverses. If there exists a path from u to v, v is said to be *reachable* from u.

For example, the sequence $\langle a, c, e, d, h \rangle$ is a path in the graph represented in Figure 16.1.

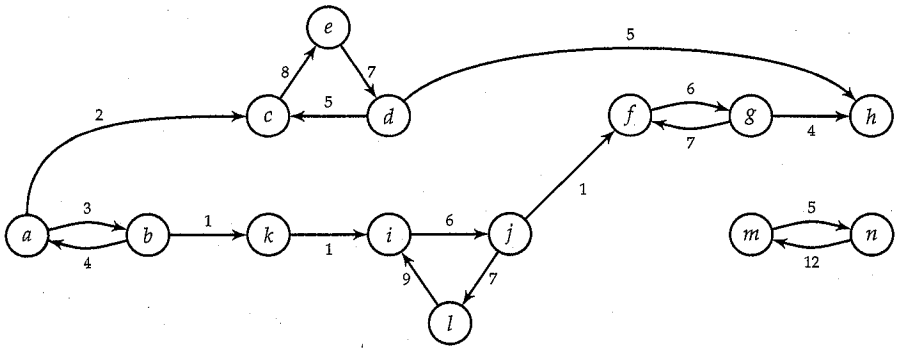


Figure 16.1: A directed graph with weights on edges.

An undirected graph is also a tuple (V, E); however E is a set of unordered pairs of V. Graphically, this is captured by drawing arrowless connections between vertices, as in Figure 16.2 on the next page.

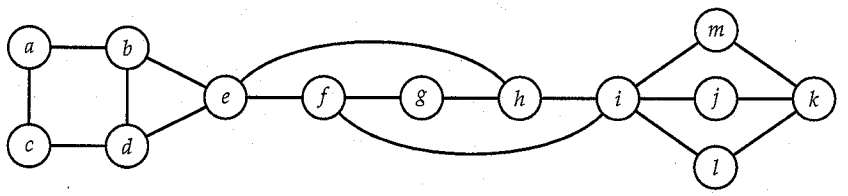


Figure 16.2: An undirected graph.

If G is an undirected graph, vertices u and v are said to be *connected* if G contains a path from u to v; otherwise, u and v are said to be *disconnected*. A graph is said to be connected if every pair of vertices in the graph is connected. A *connected component* is a maximal set of vertices C such that each pair of vertices in C is connected in G. Every vertex belongs to exactly one connected component.

A directed graph is called *weakly connected* if replacing all of its directed edges with undirected edges produces a connected undirected graph. It is *connected* if it contains a directed path from u to v or a directed path from v to u for every pair of vertices u and v. It is *strongly connected* if it contains a directed path from u to v and a directed path from v to u for every pair of vertices u and v.

Graphs naturally arise when modeling geometric problems, such as determining connected cities. However they are more general, and can be used to model many kinds of relationships.

A graph can be implemented in two ways—using *adjacency lists* or an *adjacency matrix*. In the adjacency list representation, each vertex v, has a list of vertices to which it has an edge. The adjacency matrix representation uses a $|V| \times |V|$ Boolean-valued matrix indexed by vertices, with a 1 indicating the presence of an edge. The time and space complexities of a graph algorithm are usually expressed as a function of the number of vertices and edges.

A *tree* (sometimes called a free tree) is a special sort of graph—it is an undirected graph that is connected but has no cycles. (Many equivalent definitions exist, e.g., a graph is a free tree iff there exists a unique path between every pair of vertices.) There are a number of variants on the basic idea of a tree. A rooted tree is one where a designated vertex is called the root, which leads to a parent-child relationship on the nodes. An ordered tree is a rooted tree in which each vertex has an ordering on its children. Binary trees, which are the subject of Chapter 9, differ from ordered trees since a node may have only one child in a binary tree, but that node may be a left or a right child, whereas in an ordered tree no analogous notion exists for a node with a single child. Specifically, in a binary tree, there is position as well as order associated with the children of nodes.

As an example, the graph in Figure 16.3 on the following page is a tree. Note that its edge set is a subset of the edge set of the undirected graph in Figure 16.2. Given a graph $G = (V, E)$, if the graph $G' = (V, E')$ where $E' \subset E$, is a tree, then G' is referred to as a spanning tree of G.

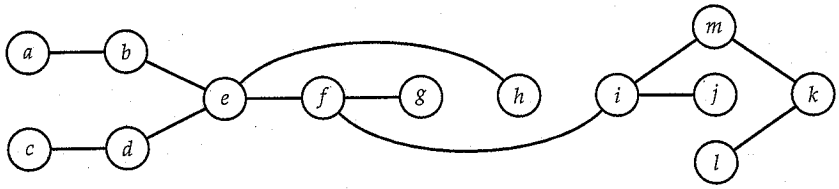


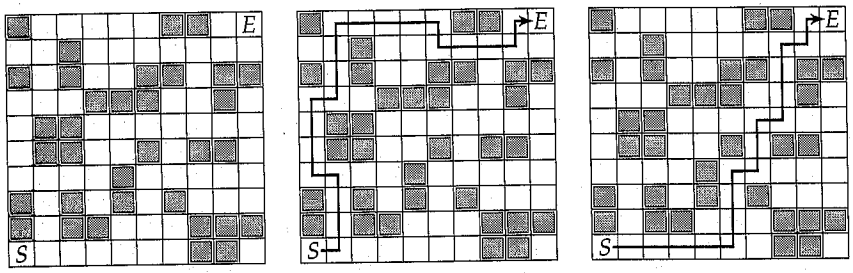
Figure 16.3: A tree.

Graph search

Computing vertices which are reachable from other vertices is a fundamental operation which can be performed by depth-first search (DFS) and breadth-first search (BFS). Both are linear time—$O(|V| + |E|)$. They differ from each other in terms of the additional information they provide, e.g., BFS can be used to compute distances from the start vertex and DFS can be used to check for the presence of cycles. Key notions in DFS include the concept of *discovery time* and *finishing time* for vertices.

16.1 Searching a maze

It is natural to apply graph models and algorithms to spatial problems. Consider a black and white digitized image of a maze—white pixels represent open areas and black spaces are walls. There are two special white pixels: one is designated the entrance and the other is the exit. The goal in this problem is to find a way of getting from the entrance to the exit, as illustrated in Figure 16.4.



(a) A maze. (b) A path from entrance to exit. (c) A shortest path from entrance to exit.

Figure 16.4: An instance of the maze search problem, with two solutions, where S and E denote the entrance and exit, respectively.

Problem 16.1: Given a 2D array of black and white entries representing a maze with designated entrance and exit points, find a path from the entrance to the exit, if one exists. *pg. 374*

16.2 Transform one string to another (☠)

Let s and t be strings and D a dictionary, i.e., a set of strings. Define s to *produce* t if there exists a sequence of strings $\sigma = \langle s_0, s_1, \ldots, s_{n-1} \rangle$ such that $s_0 = s$, $s_{n-1} = t$, for all $i, s_i \in D$, and adjacent strings have the same length and differ in exactly one character. The sequence σ is called a *production sequence*.

Problem 16.2: Given a dictionary D and two strings s and t, write a function to determine if s produces t. Assume that all characters are lowercase alphabets. If s does produce t, output the length of a shortest production sequence; otherwise, output -1. *pg. 375*

16.3 Wiring a printed circuit board

Consider a collection of electrical pins on a printed circuit board (PCB). For each pair of pins, there may or may not be a wire joining them. This is shown in Figure 16.5, where vertices correspond to pins, and edges indicate the presence of a wire between pins. (The significance of the colors is explained later.)

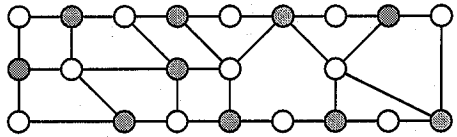


Figure 16.5: A set of pins and wires between them.

Problem 16.3: Design an algorithm that takes a set of pins and a set of wires connecting pairs of pins, and determines if it is possible to place some pins on the left half of a PCB, and the remainder on the right half, such that each wire is between left and right halves. Return such a division, if one exists. For example, the light vertices and dark vertices in Figure 16.5 are such division. *pg. 376*

16.4 Degrees of connectedness (☠)

A *connected graph* is one in which for any two vertices u and v there exists a path from u to v. The notion of connectedness holds for both directed and undirected graphs—for undirected graphs, we sometimes simply say there exists a path between u and v.

Intuitively, some graphs are more connected than others—e.g., a clique (an undirected graph in which every two vertices are connected by an edge) is more connected than a tree. To be more quantitative, we could refer to a graph as being $2\forall$-connected if it remains connected even if any single edge is removed. A graph is $2\exists$-connected if there exists an edge that can be removed while still leaving the graph connected.

The undirected graph in Figure 16.2 on Page 131 is $2\forall$-connected, since any single edge can be removed, and there will still exist a path from any vertex to any other vertex. However, if the edge (h, i) is removed, then the remaining graph, though

connected, is not 2∀-connected, since the subsequent removal of edge (f, i) results in an unconnected graph. For example, there will be no path from a to m, since all paths in the original graph pass through either (h, i) or (f, i).

The undirected graph in Figure 16.3 on Page 132 is not 2∃-connected. However adding any edge to the graph makes it 2∃-connected.

One application of this idea is in fault tolerance for data networks. Suppose you are given a set of data centers connected through a set of dedicated point-to-point links. You want to reach from any data center to any other data center through a combination of these dedicated links. Sometimes one of these links can become temporarily out of service and you want to ensure that your network can sustain up to one faulty link. How can you verify this?

Problem 16.4: Let $G = (V, E)$ be a connected undirected graph. How would you efficiently check if G is 2∃-connected? Can you make your algorithm run in $O(|V|)$ time? How would you check if G is 2∀-connected? *pg. 378*

16.5 Extended contacts

A social network consists of a set of individuals and, for each individual, a list of his contacts. (The contact relationship may not be symmetric—A may be a contact of B but B may not be a contact of A.) Define C to be an extended contact of A if he is either a contact of A or a contact of an extended contact of A.

Problem 16.5: Devise an efficient algorithm which takes a social network and computes for each individual his extended contacts. *pg. 380*

16.6 Theory of equality

Programs are usually checked using testing—a number of manually written or random test cases are applied to the program and the program's results are checked by assertions or visual inspection.

Formal verification consists of examining a program and analytically determining if there exists an input for which an assertion fails. Formal verification of programs is undecidable. However there are significant subclasses of programs for which the verification problem is decidable.

Consider the following problem. Given a set of variables $x_1, \ldots, x_n$, equality constraints of the form $x_i = x_j$, and inequality constraints of the form $x_i \neq x_j$, is it possible to satisfy all the constraints simultaneously? For example, the constraints $x_1 = x_2, x_2 = x_3, x_3 = x_4$, and $x_1 \neq x_4$ cannot be satisfied simultaneously. Such constraints arise in checking the equivalence of loop-free programs with uninterpreted functions.

Problem 16.6: Design an efficient algorithm that takes as input a collection of equality and inequality constraints and decides whether the constraints can be satisfied simultaneously. *pg. 381*

Advanced graph algorithms

Up to this point we looked at basic search and combinatorial properties of graphs. The algorithms we considered were all linear time complexity and relatively straightforward—the major challenge was in modeling the problem appropriately.

Four classes of problems on graphs can be solved efficiently, i.e., in polynomial time. Most other problems on graphs are either variants of these or, very likely, not solvable by polynomial time algorithms. These four classes are:

- Shortest paths—given a graph, directed or undirected, with costs on the edges, find the minimum cost path from a given vertex to all vertices. Variants include computing the shortest paths for all pairs of vertices, and the case where costs are all nonnegative.
- Minimum spanning tree—given a connected undirected graph $G = (V, E)$ with weights on each edge, find a subset E' of the edges with minimum total weight such that the subgraph $G' = (V, E')$ is connected.
- Matching—given an undirected graph, find a maximum collection of edges subject to the constraint that every vertex is incident to at most one edge. The matching problem for bipartite graphs is especially common and the algorithm for this problem is much simpler than for the general case. A common variant is the maximum weighted matching problem in which edges have weights and a maximum weight edge set is sought, subject to the matching constraint.
- Maximum flow—given a directed graph with a capacity for each edge, find the maximum flow from a given source to a given sink, where a flow is a function mapping edges to numbers satisfying conservation (flow into a vertex equals the flow out of it) and the edge capacities. The minimum cost circulation problem generalizes the maximum flow problem by adding lower bounds on edge capacities, and for each edge, a cost per unit flow.

In this chapter we restrict our attention to shortest-path and minimum spanning tree problems: these are subjects which anyone interviewing for a software position should be familiar with. If you have specialized knowledge of optimization or graph theory you may be asked a problem whose solution uses matching or maximum flow. We have several representative examples in Chapter 21. Specifically, maximum matching is illustrated by Problems 21.19 to 21.22 on Page 168; maximum flow is the subject of Problems 21.23 and 21.24 on Page 169. For such problems, you will most likely be asked to find the right embedding, rather than writing explicit code.

16.7 Team photo day—2

Problem 16.7: How would you generalize your solution to Problem 13.6 on Page 100. to determine the largest number of teams that can be photographed simultaneously subject to the same constraints? *pg. 382*

16.8 Minimum delay schedule, unlimited resources

Let $\mathcal{T} = \{T_0, T_1, \ldots, T_{n-1}\}$ be a set of tasks. Each task runs on a single generic server. Task T_i has a duration τ_i, and a set P_i (possibly empty) of tasks that must be completed before T_i can be started. The set is *feasible* if there does not exist a sequence of tasks $\langle T_0, T_1, \ldots, T_{n-1}, T_0 \rangle$ starting and ending at the same task such that for each consecutive pair of tasks in the sequence, the first task must be completed before the second task can begin.

Problem 16.8: Given an instance of the task scheduling problem, compute the least amount of time in which all the tasks can be performed, assuming an unlimited number of servers. Explicitly check that the system is feasible. *pg. 383*

16.9 Shortest path with fewest edges

In the usual formulation of the shortest path problem, the number of edges in the path is not a consideration. For example, considering the shortest path problem from a to h in Figure 16.1 on Page 130, the sum of the edge costs on the path $\langle a, c, e, d, h \rangle$ is 22, which is the same as for path $\langle a, b, k, i, j, f, g, h \rangle$. Both are shortest paths, but the latter has three more edges.

Heuristically, if we did want to avoid paths with a large number of edges, we can add a small amount to the cost of each edge. However depending on the structure of the graph and the edge costs, this may not result in the shortest path.

Problem 16.9: Design an algorithm which takes as input a graph $G = (V, E)$, directed or undirected, a nonnegative cost function on E, and vertices s and t; your algorithm should output a path with the fewest edges amongst all shortest paths from s to t. *pg. 384*

16.10 Quickest route

A flight is specified as a four-tuple: start-time, originating city, destination city, and arrival-time (possibly on a later day). A time-table is a set of flights. Flights are assumed to be daily.

Problem 16.10: Given a time-table, a starting city, a starting time, and a destination city, how would you compute the soonest you could get to the destination city? Assume all flights start and end on time, and that you need 60 minutes between flights. *pg. 385*

16.11 Road network

The Texas Department of Transportation is considering adding a new section of highway to the Texas Highway System. Each highway section connects two cities. City officials have submitted proposals for the new highway—each proposal includes the pair of cities being connected and the length of the section.

Problem 16.11: Devise an efficient algorithm which takes the existing highway network (specified as a set of highway sections between pairs of cities) and proposals

for new highway sections, and returns a proposed highway section which minimizes the shortest driving distance between El Paso and Corpus Christi. *pg. 386*

16.12 Arbitrage (☺)

You are exploring the remote valleys of Papua New Guinea, one of the last uncharted places in the world. You come across a tribe that does not have money—instead it relies on the barter system. A total of n commodities are traded and the exchange rates are specified by a 2D array. For example, three sheep can be exchanged for seven goats and four goats can be exchanged for 200 pounds of wheat.

Transaction costs are zero, exchange rates do not fluctuate, fractional quantities of items can be sold, and the exchange rate between each pair of commodities is finite. Table 4.4 on Page 41 shows exchange rates for currency trades, which is similar in spirit to the current problem.

Problem 16.12: Design an efficient algorithm to determine whether there exists an arbitrage—a way to start with a single unit of some commodity C and convert it back to more than one unit of C through a sequence of exchanges. *pg. 387*

16.13 Updating a minimum spanning tree

Problem 16.13: Let $G = (V, E)$ be an undirected graph with edge weight function $w : E \mapsto \mathcal{Z}^+$. You are given $T \subset E$, an MST of G. Let e be an edge. Design efficient algorithms for computing the MST when (1.) $w(e)$ decreases , and (2.) $w(e)$ increases. *pg. 389*

CHAPTER

17 Intractability

All of the general methods presently known for computing the chromatic number of a graph, deciding whether a graph has a Hamiltonian cycle, or solving a system of linear inequalities in which the variables are constrained to be 0 or 1, require a combinatorial search for which the worst-case time requirement grows exponentially with the length of the input.

— *"Reducibility Among Combinatorial Problems,"*
R. M. KARP, 1972

In real-world settings you will sometimes encounter problems that can be directly solved using efficient textbook algorithms such as binary search and shortest paths. As we have seen in the earlier chapters, it is often difficult to identify such problems because the core algorithmic problem is obscured by details. More generally, you may encounter problems which can be transformed into equivalent problems that have an efficient textbook algorithm, or problems that can be solved efficiently using meta-algorithms such as DP.

Often the problem you are given is intractable—i.e., there may not exist an efficient algorithm for the problem. Complexity theory addresses these problems. Some have been proved to not have an efficient solution (such as checking the validity of relationships involving $\exists, +, <, \Rightarrow$ on the integers) but the vast majority are only conjectured to be intractable. The conjunctive normal form satisfiability (CNF-SAT) problem (Problem 17.11 on Page 143) is an example of a problem that is conjectured to be intractable. Specifically, the CNF-SAT problem belongs to the complexity class NP—problems for which a candidate solution can be efficiently checked—and is conjectured to be the hardest problem in this class.

When faced with a problem P that appears to be intractable, the first thing to do is to prove intractability. This is usually done by taking a problem which is known to be intractable and showing how it can be efficiently reduced to P. Often this reduction gives insight into the cause of intractability.

Unless you are a complexity theorist, proving a problem to be intractable is a starting point, not an end point. Remember something is a problem only if it has a solution. There are a number of approaches to solving intractable problems:

- Brute-force solutions which are typically exponential but may be acceptable, if the instances encountered are small.
- Branch and bound techniques which prune much of the complexity of a brute-force search.
- Approximation algorithms which return a solution that is provably close to optimum.

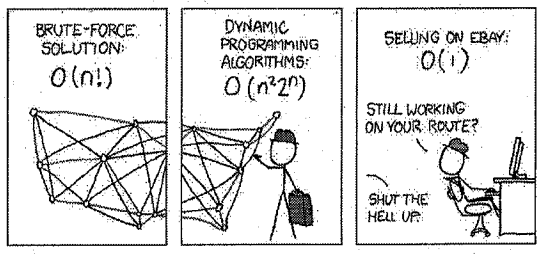


Figure 17.1: $P = NP$, by XKCD.

- Heuristics based on insight, common case analysis, and careful tuning that may solve the problem reasonably well.
- Parallel algorithms, wherein a large number of computers can work on subparts simultaneously.

Don't forget it may be possible to dramatically change the problem formulation while still achieving the higher level goal, as illustrated in Figure 17.1.

17.1 Ties in a presidential election

The US President is elected by the members of the Electoral College. The number of electors per state and Washington, D.C., are given in Table 17.1. All electors from each state as well as Washington, D.C., cast their vote for the same candidate.

Table 17.1: Electoral college votes.

State	Electors	State	Electors	State	Electors
Alabama	9	Louisiana	8	Ohio	18
Alaska	3	Maine	4	Oklahoma	7
Arizona	11	Maryland	10	Oregon	7
Arkansas	6	Massachusetts	11	Pennsylvania	20
California	55	Michigan	16	Rhode Island	4
Colorado	9	Minnesota	10	South Carolina	9
Connecticut	7	Mississippi	6	South Dakota	3
Delaware	3	Missouri	10	Tennessee	11
Florida	29	Montana	3	Texas	38
Georgia	16	Nebraska	5	Utah	6
Hawaii	4	Nevada	6	Vermont	3
Idaho	4	New Hampshire	4	Virginia	13
Illinois	20	New Jersey	14	Washington	12
Indiana	11	New Mexico	5	West Virginia	5
Iowa	6	New York	29	Wisconsin	10
Kansas	6	North Carolina	15	Wyoming	3
Kentucky	8	North Dakota	3	Washington, D.C.	3

Problem 17.1: How would you programmatically determine if a tie is possible in a presidential election with two candidates, R and D? *pg. 389*

17.2 The knapsack problem

A thief breaks into a clock store. His knapsack will hold at most w ounces of clocks. Clock i weighs w_i ounces and retails for v_i dollars. The thief must either take or leave

a clock, and he cannot take a fractional amount of an item. His intention is to take clocks whose total value is maximum subject to the knapsack capacity constraint. His problem is illustrated in Figure 17.2. If the knapsack can hold at most 130 ounces, he cannot take all the clocks. If he greedily chooses clocks, in decreasing order of value-to-weight ratio, he will choose P, H, O, B, I, and L in that order for a total value of \$669. However, $\{H, J, O\}$ is the optimum selection, yielding a total value of \$695.

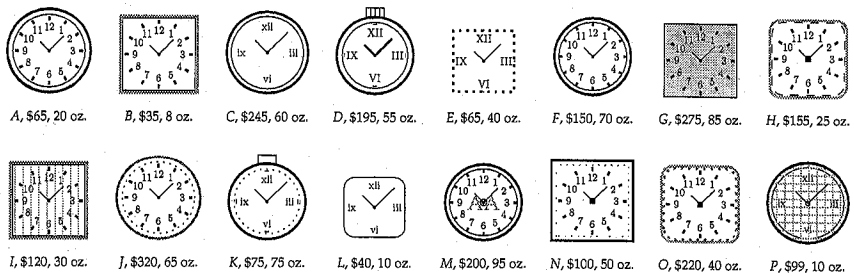


Figure 17.2: A clock store.

Problem 17.2: Design an algorithm for the knapsack problem that selects a subset of items that has maximum value and weighs at most w ounces. All items have integer weights and values. *pg. 390*

17.3 Dividing the spoils

Two thieves have successfully completed a burglary. They want to know how to divide the stolen items into two groups such that the difference between the value of these two groups is minimized. For example, they may have stolen the clocks in Figure 17.2, and would like to divide the clocks between them so that the difference of the dollar value of the two sets is minimized. For this instance, an optimum split is $\{A, G, J, M, O, P\}$ to one thief and the remaining to the other thief. The first set has value \$1179, and the second has value \$1180. An equal split is impossible, since the sum of the values of all the clocks is odd.

Problem 17.3: Let array A be an array of n positive integers. Entry $A[i]$ is the value of the i-th stolen item. Design an algorithm that computes a subset $S \subset \mathcal{Z}_n = \{0, 1, 2, \ldots, n-1\}$ such that $|\sum_{i \in S} A[i] - \sum_{j \in \mathcal{Z}_n \setminus S} A[j]|$ is minimized. *pg. 390*

17.4 Measuring with defective jugs

You have three measuring jugs, A, B, and C. The measuring marks have worn out, making it impossible to measure exact volumes. Specifically, each time you measure with A, all you can be sure of is that you have a volume that is in the range $[230, 240]$ mL. (The next time you use A, you may get a different volume—all that you know with certainty is that the quantity will be in $[230, 240]$ mL.) Jugs B and C can be used to measure a volume in $[290, 310]$ mL and in $[500, 515]$ mL, respectively. Your recipe

for chocolate chip cookies calls for at least 2100 mL and no more than 2300 mL of milk.

Problem 17.4: Write a program that determines a sequence of steps by which the required amount of milk can be obtained using the worn-out jugs. The milk is being added to a large mixing bowl, and hence cannot be removed from the bowl. Furthermore, it is not possible to pour one jug's contents into another. Your scheme should always work, i.e., return between 2100 and 2300 mL of milk, independent of how much is chosen in each individual step, as long as that quantity satisfies the given constraints. *pg. 391*

17.5 Delay-constrained shortest-path (☹)

In the conventional form of the shortest-path problem, we seek the path with the lowest cost. There exist natural situations where each edge has a cost and a delay. For example, a shipping company may have a number of locations. Sending a package along a given route incurs a cost and a delay that is the sum of the costs and delays of the individual edges on the route. This motivates the following.

Problem 17.5: Given a graph $G = (V, E)$, with cost function $c : E \mapsto \mathcal{Z}^+$, delay function $d : E \mapsto \mathcal{Z}^+$, designated vertices s and t, and a delay constraint $\Delta \in \mathcal{Z}^+$, find a path from s to t with minimum cost, subject to the constraint that the delay of the path is no more than Δ. Costs are additive—the cost of a path is the sum of the costs of the individual edges; the same holds for delays. *pg. 392*

17.6 Traveling salesman in the plane (☹)

Suppose a salesman needs to visit a set of cities $A_0, A_1, \ldots, A_{n-1}$. For any ordered pair of cities (A_i, A_j), the cost of traveling from the first to the second city is $c(A_i, A_j)$. We need to design a low cost tour for the salesman.

A tour is a sequence of cities $\langle B_0, B_1, \ldots, B_{n-1}, B_0 \rangle$. It can start at any city and the salesman can visit the cities in any order. All the cities must appear in the subsequence $\langle B_0, B_1, \ldots, B_{n-1} \rangle$. (Note that this implies that all the cities in this subsequence are distinct.) The cost of the tour is the sum of the costs of the n successive pairs $(B_i, B_{i+1 \bmod n})$, for $i = 0$ to $n - 1$.

Determining the minimum cost tour is a classic NP-hard problem and the problem remains hard even if we just ask for a tour whose cost is within a given multiple M of the minimum cost tour. However there is a special case for which this problem can be efficiently solved with a reasonable bound on the quality of the solution.

Problem 17.6: Suppose you are given a set of cities in the Cartesian plane, as shown in Figure 4.3 on Page 36. The cost of traveling from one city to another is a constant multiple of the distance between the cities. Give an efficient procedure for computing a tour whose cost is no more than two times the cost of an optimum tour. *pg. 393*

17.7 The warehouse location problem

Let $c_0, c_1, \ldots, c_{n-1}$ be n cities. We want to choose k of these cities to build warehouses in. We would like the remaining cities to be close to the warehouses. Let's say the cost of a warehouse assignment is defined to be the maximum distance of any city to a warehouse. Finding an optimum warehouse assignment is known to be NP-hard.

For example, consider the cities specified in Figure 4.3 on Page 36. Assume the coordinates for each city correspond to their positions in the Cartesian plane, and distances are the standard Euclidean distance. Then if $k = 3$, the optimum cities to place warehouses in are Los Angeles, Dakar, and Darwin. For $k = 4$, the optimum cities are Mexico City, Accra, Irkutsk, and Melbourne.

Problem 17.7: Design a fast algorithm for selecting k warehouse locations that is provably within a constant factor of the optimum solution. *pg. 394*

17.8 Sudoku solver

Sudoku is described in Problem 6.14 on Page 57. In this problem you are to write a Sudoku solver. The decision version of the generalized Sudoku problem is NP-complete; however this is restricted to the traditional 9×9 grid.

Problem 17.8: Implement a Sudoku solver. Your program should read an instance of Sudoku from the command line. The command line argument is a sequence of 3-digit strings, each encoding a row, a column, and a digit at that location. *pg. 395*

17.9 Expression synthesis

Consider an expression of the form $v_0 \odot_0 v_1 \odot_1 v_2 \cdots \odot_{n-2} v_{n-1}$. Suppose the v_is are constant integers and the $\odot_i$s are operators. The expression takes different values based on what operators we choose.

Determining an operator assignment such that the resulting expression satisfies a constraint is, in general, a difficult problem. For example, suppose the operators are $+$ and $-$, and we want to know whether we can select each $\odot_i$ such that the resulting expression evaluates to 0. The problem of partitioning a set of integers into two subsets which sum up to the same value, which is a famous NP-complete problem, directly reduces to our problem.

Problem 17.9: Given an array of digits A and a nonnegative integer k, intersperse multiplies ($\times$) and adds ($+$) with the digits of A such that the resulting arithmetical expression evaluates to k. For example, if A is $\langle 1,2,3,2,5,3,7,8,5,9 \rangle$ and k is 995, then k can be realized by the expression "$123 + 2 + 5 \times 3 \times 7 + 85 \times 9$". *pg. 398*

17.10 Computing x^n

A straight-line program for computing x^n is a finite sequence $\langle x, x^{i_1}, x^{i_2}, \ldots, x^n \rangle$ where each element after the first is either the square of some previous element or the

product of any two previous elements. For example, the term x^{15} can be computed by the straight-line programs $P1$ and $P2$:

$$\begin{aligned} P1 &= \langle x, x^2, x^4 = (x^2)^2, x^8 = (x^4)^2, x^{12} = x^8x^4, x^{14} = x^{12}x^2, x^{15} = x^{14}x \rangle \\ P2 &= \langle x, x^2, x^3 = x^2x, x^5 = x^3x^2, x^{10} = (x^5)^2, x^{15} = x^{10}x^5 \rangle \end{aligned}$$

The number of multiplications to evaluate x^n is the number of terms in the shortest such program sequence minus one. No efficient method is known for the problem of determining the minimum number of multiplications needed to evaluate x^n; the problem for multiple exponents is known to be NP-complete.

Problem 17.10: Given a positive integer n, how would you determine the minimum number of multiplications to evaluate x^n? *pg. 400*

17.11 CNF-SAT

A Boolean logic expression using logical OR (+), AND (·) and complement is said to be in conjunctive normal form (CNF) if complement is only applied to variables and the operation + is applied to variables or their negation. For example, $(a + b + c') \cdot (a' + b) \cdot (a + c' + d)$ is in CNF. The terms $a + b + c'$, $a' + b$, and $a + c' + d$ are *clauses*.

A CNF expression is said to be satisfiable if there exists an assignment of Boolean values to the variables that sets each clause to true. The assignment $a = b = d =$ `true`, $c =$ `false` is a satisfying assignment for the example given above.

Problem 17.11: Design an algorithm for checking if a CNF expression is satisfiable. *pg. 401*

17.12 Checking the Collatz conjecture

Lothar Collatz proposed this remarkable conjecture in 1937: "Define $C : \{1, 2, 3, \ldots\} \mapsto \{1, 2, 3, \ldots\}$ to be $\frac{n}{2}$ for even n, and $3n+1$, otherwise. Then for any choice of n, $C^i(n) = 1$, for some i." For example, if we start with the number 11 and iteratively compute $C^i(11)$, we get the sequence $\langle 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1 \rangle$.

Despite intense efforts, the Collatz conjecture has not been proved or disproved. Suppose you were given the task of checking the Collatz conjecture for the first billion integers. A direct approach would be to compute the convergence sequence for each number in this set.

Problem 17.12: How would you test the Collatz conjecture for the first n positive integers? *pg. 401*

17.13 Minimum delay scheduling, limited resources

Problem 17.13: You need to schedule n lectures in m classrooms. Some of those lectures are prerequisites for others. All lectures are one hour-long and start on the hour. How would you choose when and where to hold lectures to finish all the lectures as soon as possible? *pg. 402*

CHAPTER

18 Parallel Computing

The activity of a computer must include the proper reacting to a possibly great variety of messages that can be sent to it at unpredictable moments, a situation which occurs in all information systems in which a number of computers are coupled to each other.

— *"Cooperating sequential processes,"*
E. W. DIJKSTRA, 1965

Parallel computation has become increasingly common. For example, laptops and desktops come with multiple processors which communicate through shared memory. High-end computation is often done using clusters consisting of individual computers communicating through a network.

Parallelism provides a number of benefits:

- High performance—more processors working on a task (usually) means it is completed faster.
- Better use of resources—a program can execute while another waits on the disk or network.
- Fairness—letting different users or programs share a machine rather than have one program run at a time to completion.
- Convenience—it is often conceptually more straightforward to do a task using a set of concurrent programs for the subtasks rather than have a single program manage all the subtasks.
- Fault tolerance—if a machine fails in a cluster that is serving web pages, the others can take over.

Concrete applications of parallel computing include graphical user interfaces (GUI) (a dedicated thread handles UI actions resulting in increased responsiveness), Java virtual machines (a separate thread handles garbage collection which would otherwise lead to blocking), web servers (a single logical thread handles a single client request), scientific computing (a large matrix multiplication can be split across a cluster), and web search (multiple machines crawl, index, and retrieve web pages).

The two primary models for parallel computation are the shared memory model, in which each processor can access any location in memory, and the distributed memory model, in which a processor must explicitly send a message to another processor to access its memory. The former is more appropriate in the multicore setting and the latter is more accurate for a cluster. The questions in this chapter are mostly focused on the shared memory model. We cover a few problems related to

the distributed memory model, such as leader election and sorting large data sets, at the end of the chapter.

Writing correct parallel programs is challenging because of the subtle interactions between parallel components. One of the key challenges is races—two concurrent instruction sequences access the same address in memory and at least one of them writes to that address. Other challenges to correctness are

- starvation (a processor needs a resource but never gets it, e.g., Problem 18.5 on the following page),
- deadlock (Thread A acquires Lock $L1$ and Thread B acquires Lock $L2$, following which A tries to acquire $L2$ and B tries to acquire $L1$ as in Problem 18.10 on Page 148), and
- livelock (a processor keeps retrying an operation that always fails).

Bugs caused by these issues are difficult to find using testing. Debugging them is also difficult because they may not be reproducible since they are usually load dependent. It is also often true that it is not possible to realize the performance implied by parallelism—sometimes a critical task cannot be parallelized, making it impossible to improve performance, regardless of the number of processors added. Similarly, the overhead of communicating intermediate results between processors can exceed the performance benefits.

18.1 Service with caching

Problem 18.1: Design an online spell correction system. It should take as input a string s and return an array of entries in its dictionary which are closest to the string using the Levenshtein distance specified in Problem 15.11 on Page 120. Cache the most recently computed result. *pg. 403*

18.2 Thread pools

The following class, `SimpleWebServer`, implements part of a simple HTTP server:

```
public class SimpleWebServer {
  final static int PORT = 8080;
  public static void main (String [] args) throws IOException {
    ServerSocket serversock = new ServerSocket(PORT);
    for (;;) {
      Socket sock = serversock.accept();
      ProcessReq(sock);
    }
  }
}
```

Problem 18.2: Suppose you find that the `SimpleWebServer` has poor performance because `processReq` frequently blocks on I/O. What steps could you take to improve `SimpleWebServer`'s performance? *pg. 405*

18.3 Asynchronous callbacks

It is common in a distributed computing environment for the responses to not return in the same order as the requests were made. One way to handle this is through an "asynchronous callback"—a method to be invoked on response. This is formalized by a `Requester` class.

A `Requester class` implements a `Dispatch` method which takes a `Requester` object. The `Requester` object includes a `request` string, a `ProcessResponse(string response)` method, and an `Execute` method that takes a string and returns a string. `Dispatch` is to create a new thread which invokes `Execute` on `request`. When `Execute` returns, `Dispatch` invokes the `ProcessResponse` method on the response.

Problem 18.3: Implement a `Requester` class. The `Execute` method may take an indeterminate amount of time to return; it may never return. You need to have a time-out mechanism for this. Assume `Requester` objects have an `Error` method that you can invoke. *pg. 406*

18.4 Timer

Consider a web-based calendar in which the server hosting the calendar has to perform a task when the next calendar event takes place. (The task could be sending an email or a Short Message Service (SMS).) Your job is to design a facility that manages the execution of such tasks.

Problem 18.4: Develop a `Timer` class that manages the execution of deferred tasks. The `Timer` constructor takes as its argument an object which includes a `Run` method and a `name` field, which is a string. `Timer` must support—(1.) starting a thread, identified by name, at a given time in the future; and (2.) canceling a thread, identified by name (the cancel request is to be ignored if the thread has already started). *pg. 407*

18.5 Readers-writers

Consider an object s which is read from and written to by many threads. (For example, s could be the cache from Problem 18.1 on the previous page.) You need to ensure that no thread may access s for reading or writing while another thread is writing to s. (Two or more readers may access s at the same time.)

One way to achieve this is by protecting s with a mutex that ensures that two threads cannot access s at the same time. However this solution is suboptimal because it is possible that a reader $R1$ has locked s and another reader $R2$ wants to access s. Reader $R2$ does not have to wait until $R1$ is done reading; instead, $R2$ should start reading right away.

This motivates the first readers-writers problem: protect s with the added constraint that no reader is to be kept waiting if s is currently opened for reading.

Problem 18.5: Implement a synchronization mechanism for the first readers-writers problem. *pg. 407*

18.6 Readers-writers with write preference

Suppose we have an object s as in Problem 18.5 on the facing page. In the solution to Problem 18.5 on the preceding page, a reader $R1$ may have the lock; if a writer W is waiting for the lock and then a reader $R2$ requests access, $R2$ will be given priority over W. If this happens often enough, W will starve. Instead, suppose we want W to start as soon as possible.

This motivates the second readers-writers problem: protect s with "writer-preference", i.e., no writer, once added to the queue, is to be kept waiting longer than absolutely necessary.

Problem 18.6: Implement a synchronization mechanism for the second readers-writers problem. *pg. 408*

18.7 Readers-writers with fairness

The specifications to both Problems 18.5 on the facing page and 18.6 can lead to starvation—the first may starve writers and the second may starve readers. The third readers-writers problem adds the constraint that neither readers nor writers should starve.

Problem 18.7: Implement a synchronization mechanism for the third readers-writers problem. *pg. 409*

18.8 Producer-consumer queue

Two threads, the producer P and the consumer C, share a fixed length array of strings A. The producer generates strings one at a time which it writes into A; the consumer removes strings from A, one at a time.

Problem 18.8: Design a synchronization mechanism for A which ensures that P does not try to add a string into the array if it is full and C does not try to remove data from an empty buffer. *pg. 409*

18.9 Barber shop

Consider a barber shop with a single barber B, one barber chair, and n chairs for customers who are waiting for their turn for a haircut. The barber sleeps in his chair when customers are not present. On entering, a customer either awakens the barber or if the barber is cutting someone else's hair, he sits down in one of the chairs for waiting customers. If all of the waiting chairs are taken, the newly arrived customer simply leaves.

Problem 18.9: Model the barber shop using semaphores and mutexes to ensure correct behavior. Each customer is a thread, as is the barber. *pg. 409*

18.10 Dining philosophers

In the dining philosophers problem n threads, numbered from 0 to $n-1$, run concurrently. Resources are numbered from 0 to $n-1$. Thread i requires resources i and $i+1 \bmod n$ before it can invoke a method m. (The problem gets its name because it models n philosophers sitting at a round table, alternating between thinking, eating, and waiting. A single chopstick is present between each pair of philosophers. To eat, a philosopher must hold two chopsticks—one placed immediately to his left and one immediately to his right.)

Problem 18.10: Implement a synchronization mechanism for the dining philosophers problem. *pg. 410*

18.11 Checking the Collatz conjecture in parallel

In Problem 17.12 on Page 143 and its solution we introduced the Collatz conjecture and heuristics for checking it. In this problem, you are to build a parallel checker for the Collatz conjecture. Specifically, assume your program will run on a multicore machine, and threads in your program will be distributed across the cores. Your program should check the Collatz conjecture for every integer in $[1, U]$ where U is an input to your program.

Problem 18.11: Design a multi-threaded program for checking the Collatz conjecture. Make full use of the cores available to you. To keep your program from overloading the system, you should not have more than n threads running at a time. *pg. 410*

18.12 Broadcast in a rooted tree

Consider a computer network organized as a rooted tree. A node can send a message to only one child at a time, and it takes one second for the child to receive the message. The root periodically receives a message from an external source. It needs to send this message to all the nodes in the tree. The root has complete knowledge of how the network is organized.

Problem 18.12: Design an algorithm that computes the sequence of transfers that minimizes the time taken to transfer a message from the root to all the nodes in the tree. *pg. 412*

18.13 Leader election

You are to devise a protocol by which a collection of hosts on the Internet can elect a leader. Hosts can communicate with each other using Transmission Control Protocol (TCP) connections. For host A to communicate with host B, it needs to know B's IP address. Each host starts off with a set of IP addresses and the protocol code that you implement will run on all the hosts on a fixed port.

Problem 18.13: Devise a protocol by which hosts can elect a leader from the set of all hosts participating in the protocol. The protocol should be fast, in that it

converges quickly; it should be efficient, in that it should use few connections and small messages. *pg. 412*

18.14 TeraSort and PetaSort

Modern datasets are huge. For example, it is estimated that a popular social networking website contains over two trillion distinct items.

Problem 18.14: How would you sort a billion 1000 byte strings? How about a trillion 1000 byte strings? *pg. 414*

18.15 Distributed throttling

You have n machines ("crawlers") for downloading the entire web. The responsibility for a given URL is assigned to the crawler whose ID is Hash(URL) mod n. Downloading a web page takes away bandwidth from the web server hosting it.

Problem 18.15: Implement crawling under the constraint that in any given minute your crawlers do not request more than b bytes from any website. *pg. 415*

CHAPTER

19

Design Problems

We have described a simple but very powerful and flexible protocol which provides for variation in individual network packet sizes, transmission failures, sequencing, flow control, and the creation and destruction of process- to-process associations.

— *"A Protocol for Packet Network Intercommunication,"*
V. G. CERF AND R. E. KAHN, 1974

This chapter is concerned with system design problems. These problems are fairly open-ended, and many can be the starting point for a large software project or a Ph.D. A comprehensive discussion on the solutions available for such problems is outside the scope of this book. In an interview setting when someone asks such a question, you should have a conversation in which you demonstrate an ability to think creatively, understand design trade-offs, and attack unfamiliar problems. You should sketch key data structures and algorithms, as well as the technology stack (programming language, libraries, OS, hardware, and services) that you would use to solve the problem. Some specific things to keep in mind are implementation time, scalability, testability, security, and internationalization.

The answers in this chapter are presented in this context—they are meant to be examples of good responses in an interview and are not definitive state-of-the-art solutions.

19.1 CREATING PHOTOMOSAICS

A photomosaic is built from a collection of images called "tiles" and a target image. The photomosaic is another image which approximates the target image and is built by juxtaposing the tiles. The quality of approximation is defined by human perception.

Problem 19.1: Design a program that produces high quality mosaics with minimal compute time. *pg. 415*

19.2 SEARCH ENGINE

Keyword-based search engines maintain a collection of several billion documents. One of the key computations performed by a search engine is to retrieve all the documents that contain the keywords contained in a query. This is a nontrivial task in part because it must be performed in a few tens of milliseconds.

Here we consider a smaller version of the problem where the collection of documents can fit within the RAM of a single computer.

Problem 19.2: Given a million documents with an average size of 10 kilobytes, design a program that can efficiently return the subset of documents containing a given set of words. *pg. 416*

19.3 IP FORWARDING (☺)

In many applications instead of an exact match of strings, we are looking for a prefix match, i.e., given a set of strings and a search string, we want to find longest string from the set that is a prefix of the search string. One application is the IP route lookup problem. When an IP packet arrives at a router, the router looks up the next hop for the packet by searching the destination IP address of the packet in its routing table. The routing table is specified as a set of prefixes of IP addresses and the router has to identify the longest matching prefix. If this task is to be done only once, it is impossible to do better than testing each prefix in the routing table. However an Internet core router does millions of lookups on destination addresses over the set of prefixes each second. Therefore it is advantageous to do some precomputation.

Problem 19.3: You are given a large set of strings S. Given a query string Q, how would you design a system that can quickly identify the longest string $p \in S$ that is a prefix of Q? *pg. 417*

19.4 SPELL CHECKER

Designing a good spelling correction system can be challenging. We discussed spelling correction in the context of edit distance (Problem 15.11 on Page 120). However in that problem, we only computed the Levenshtein distance between a pair of strings. A spell checker must find a set of words that are closest to a given word from the entire dictionary. Furthermore, the Levenshtein distance may not be the right distance function when performing spelling correction—it does not take into account the commonly misspelled words or the proximity of letters on a keyboard.

Problem 19.4: How would you build a spelling correction system? *pg. 417*

19.5 STEMMING

When a user submits the query "computation" to a search engine, it is quite possible he might be interested in documents containing the words "computers", "compute", and "computing" also. If you have several keywords in a query, it becomes difficult to search for all combinations of all variants of the words in the query.

Stemming is the process of reducing all variants of a given word to one common root, both in the query string and in the documents. An example of stemming would be mapping {*computers, computer, compute*} to *compute*. It is almost impossible to succinctly capture all possible variants of all words in the English language but a few simple rules can get us most cases.

Problem 19.5: Design a stemming algorithm that is fast and effective. *pg. 418*

19.6 TEX

The TEX system for typesetting beautiful documents was designed by Don Knuth. Unlike GUI based document editing programs, TEX relies on a markup language, which is compiled into a device independent intermediate representation. TEX formats text, lists, tables, and embedded figures; supports a very rich set of fonts and mathematical symbols; automates section numbering, cross-referencing, index generation; exports an API; and much more.

Problem 19.6: How would you implement TEX? *pg. 418*

19.7 UNIX TAIL

The UNIX `tail` command displays the last part of a file. For this problem, assume that `tail` takes two arguments—a filename, and the number of lines, starting from the last line, that are to be printed.

Problem 19.7: Implement the UNIX `tail` command. *pg. 419*

19.8 COPYRIGHT INFRINGEMENT

YouTV.com is a successful online video sharing site. Hollywood studios complain that much of the material uploaded to the site violates copyright.

Problem 19.8: Design a feature that allows a studio to enter a set V of videos that belong to it, and to determine which videos in the YouTV.com database match videos in V. *pg. 420*

19.9 IMPLEMENT PAGERANK

The PageRank algorithm assigns a rank to a web page based on the number of "important" pages that link to it. The algorithm essentially amounts to the following:

1. Build a matrix A based on the hyperlink structure of the web. Specifically, $A_{ij} = \frac{1}{d_i}$ if page i links to page j; here d_i is the total number of unique outgoing links from page i.
2. Solve for X satisfying $X = \epsilon[1] + (1-\epsilon)A^T X$. Here ϵ is a scalar constant (e.g., $\frac{1}{7}$) and $[1]$ represents a column vector of 1s. The value $X[i]$ is the rank of the i-th page.

The most commonly used approach to solving the above equation is to start with a value of X, where each component is $\frac{1}{n}$ (where n is the number of pages) and then perform the following iteration: $X_k = \epsilon[1] + (1-\epsilon)A^T X_{k-1}$.

Problem 19.9: Design a system that can compute the ranks of ten billion web pages in a reasonable amount of time. *pg. 421*

19.10 SCALABLE PRIORITY SYSTEM

Maintaining a set of prioritized jobs in a distributed system can be tricky. Applications include a search engine crawling web pages in some prioritized order, as well

as event-driven simulation in molecular dynamics. In both cases the number of jobs is in the billions and each has its own priority.

Problem 19.10: Design a system for maintaining a set of prioritized jobs that implements the following API:

1. Insert a new job with a given priority.
2. Delete a job.
3. Find the highest priority job.

Each job has a unique ID. Assume the set cannot fit into a single machine's memory.

pg. 421

19.11 Latency reduction

The Pareto distribution is:

$$P[X > x] = \begin{cases} 1 - \left(\frac{x_m}{x}\right)^{\alpha}, & \text{if } x > x_m; \\ 1, & \text{otherwise.} \end{cases}$$

Here α and x_m are parameters of the distribution. It is one of the heavy-tailed distributions that commonly occur in various workloads.

Imagine you are running a service on k servers and that any service request can be processed by any of the servers. A given server can process only one request at a time. A server takes $t(r)$ time to process request r, and $t(r)$ follows a Pareto distribution.

Clients of a service often care more about the 99-th or the 95-th percentile latency for a request to be served than the average latency since they want most of the requests to be serviced in a reasonable amount of time even if a request occasionally takes a long time.

Problem 19.11: You have guaranteed your clients that 99% of their requests will be serviced in less than one second. How would you design a system to meet this requirement with minimal cost?

pg. 422

19.12 Online advertising system

Problem 19.12: Jingle, a search engine startup, wants to monetize its search results by displaying advertisements alongside search results. Design an online advertising system for Jingle.

pg. 422

19.13 Recommendation system

Jingle wants to generate more page views on its news site. A product manager has the idea to add to each article a sidebar of clickable snippets from articles that are likely to be of interest to someone reading the current article.

Problem 19.13: Design a system that automatically generates a sidebar of related articles.

pg. 423

19.14 Driving directions

As a part of its charter to collect all the information in the world and make it universally accessible, Jingle wants to develop a driving directions service. Users enter a start and end address; the service returns directions.

Problem 19.14: Design a driving directions service with a web interface. *pg. 424*

19.15 Distributing large files

Jingle is developing a search feature for breaking news. New articles are collected from a variety of online news sources such as newspapers, bulletin boards, and blogs, by a single lab machine at Jingle. Every minute, roughly one thousand articles are posted and each article is 100 kilobytes.

Jingle would like to serve these articles from a data center consisting of a 1000 servers. For performance reasons, each server should have its own copy of articles that were recently added. The data center is far away from the lab machine.

Problem 19.15: Design an efficient way of copying one thousand files each 100 kilobytes in size from a single lab server to each of 1000 servers in a distant data center. *pg. 425*

19.16 Online poker

Clump Enterprises wants to create a website by which gamblers can play poker online.

Problem 19.16: Design an online poker playing service for Clump Enterprises. Describe both the system architecture and a set of classes. *pg. 425*

19.17 Design the World Wide Web

Problem 19.17: Design the World Wide Web. Specifically, describe what happens when you enter a URL in a browser address bar, and press return. *pg. 426*

CHAPTER

20

Probability

The theory of probability, as a mathematical discipline, can and should be developed from axioms in exactly the same way as Geometry and Algebra.

— *"Foundations of the Theory of Probability,"*
A. N. KOLMOGOROV, 1933

Probability comes up often in computation, e.g., when modeling random events (input data and arrival time), and designing efficient algorithms, quicksort and selecting the k-th element being notable examples. It is a rich subject and is the source of many interview questions.

To a first approximation, a probability measure is a function p from subsets of a set E of events to $[0,1]$ that has the properties that $p(E) = 1$ and $p(A \cup B) = p(A) + p(B)$ when A and B are disjoint. Various properties and notations can be given around these concepts. For example, it is easy to prove that $p(A \cup B) = p(A) + p(B) - p(A \cap B)$ for general A and B.

A random variable X is a function from E to $(-\infty, \infty)$; it can be characterized by a *cumulative distribution function* $F_X : \Re \mapsto [0,1]$, where $F_X(\tau) = p(X^{-1}((-\infty, \tau]))$. When X takes a finite or countable set of values, we can talk about the probability of X taking a particular value, i.e., $p(X = \tau_i)$. If X takes a continuous range of values and F_X is differentiable, we talk of $f_X(\tau) = \frac{dF_X}{d\tau}$ as being the *probability density function*.

The expected value $E[X]$ of a random variable X taking a finite set of values $T = \{\tau_0, \tau_1, \ldots, \tau_{n-1}\}$ is simply $\sum_{\tau_i \in T} \tau_i \cdot p(X = \tau_i)$, i.e., it is the average value of X, weighted by probabilities. The notion of expectation generalizes to countable sets of values. For a random variable taking a continuous set of values, the sum is replaced with an integral and the weighting function is the probability density function. The variance $\mathrm{var}(X)$ of a random variable X is the expected value of $(X - E[X])^2$, and, in some sense, measures how spread out the variable is. Some of the key results in probability have to do with bounds on the probability of events, e.g., the Chebyshev bound: $\Pr\left(|X - E[X]| \geq k\sqrt{\mathrm{var}(X)}\right) \leq \frac{1}{k^2}$ holds for arbitrary distributions.

The following random variables are frequently encountered. The Bernoulli random variable takes only two values, 0 and 1; it is used, for example, in modeling coin tosses. The uniform random variable takes values in an interval U; the probability of $I \subset U$ is proportional to the length of I. The Poisson random variable takes non-negative values—it models the number of events in a fixed period of time, e.g., the number of HTTP requests in a minute. The Gaussian random variable takes all real values. Let $X_0, X_1, X_2, \ldots$ be independent identically distributed random variables

each with mean μ and variance σ^2. Then $(\sum_{i=0}^{n-1}(X_i - \mu))/\sqrt{n}$ tends to a zero mean Gaussian random variable with unit variance.

For the most part, probability is intuitive. However, there are notable exceptions. For example, at first glance, it would seem impossible for there to exist three 6-sided dice $A, B,$ and C such that A is more likely to roll a higher number than B, B is more likely to roll a higher number than C, and C is more likely to roll a higher number than A. However if A has sides $2, 2, 4, 4, 9,$ and 9, B has sides $1, 1, 6, 6, 8,$ and 8, and die C has sides $3, 3, 5, 5, 7,$ and 7, then the probability that A rolls a higher number than B is $\frac{20}{36}$, the probability that B rolls a higher number than C is $\frac{20}{36}$, and the probability that C rolls a higher number than A is $\frac{20}{36}$. The Monty Hall problem is another famous example.

20.1 Random permutations—1

Consider estimating the probability of winning a game of Blackjack, assuming cards are shuffled perfectly uniformly before dealing hands and everyone is playing according to a known strategy. One way to do this would be to generate a few random permutations of the cards and compute the chances of winning in each case. It is important that the process used to generate a random permutation generates each permutation with equal probability.

Problem 20.1: Does the following process yield a uniformly random permutation of A? "For $i \in \{0, 1, \ldots, n-1\}$, swap $A[i]$ with a randomly chosen element of A." (The randomly chosen element could be i itself.) *pg. 427*

20.2 Offline sampling

Problem 20.2: Let A be an array of n distinct elements. Design an algorithm that returns a subset of k elements of A. All subsets should be equally likely. Use as few calls to the random number generator as possible and use $O(1)$ additional storage. You can return the result in the same array as input. *pg. 427*

20.3 Random permutations—2

Problem 20.1 showed that generating random permutations is not as straightforward as it seems.

Problem 20.3: Design an algorithm that creates uniformly random permutations of $\{0, 1, \ldots, n-1\}$. You are given a random number generator that returns integers in the set $\{0, 1, \ldots, n-1\}$ with equal probability; use as few calls to it as possible. *pg. 429*

20.4 Uniform random number generation

The next problem is motivated by the following scenario. Five friends have to select a designated driver using a single unbiased coin. The process should be fair to everyone.

Problem 20.4: How would you implement a random number generator that generates a random integer i in $[a, b]$, given a random number generator that produces either zero or one with equal probability? All generated values should have equal probability. What is the run time of your algorithm, assuming each call to the given random number generator takes $O(1)$ time? *pg. 429*

20.5 Nonuniform random number generation

Suppose you need to write a load test for a server. You have studied the inter-arrival time of requests to the server over a period of one year and from this data have computed a histogram of the distribution of the inter-arrival time of requests. In the load test you would like to generate requests for the server such that the inter-arrival times come from the same distribution that was observed in the historical data. The following problem formalizes the generation of inter-arrival times.

Problem 20.5: You are given a set T of n nonnegative real numbers $\{t_0, t_1, \ldots, t_{n-1}\}$ and probabilities $p_0, p_1, \ldots, p_{n-1}$, where $\sum_{i=0}^{n-1} p_i = 1$. Assume that $t_0 < t_1 < \cdots < t_{n-1}$. Given a random number generator that produces values in $[0, 1]$ uniformly, how would you generate a value X from T according to the specified probabilities? *pg. 430*

20.6 Reservoir sampling

The following problem is motivated by the design of a packet sniffer that provides a uniform sample of packets for a network session.

Problem 20.6: Design an algorithm that reads a sequence of packets and maintains a uniform random subset of size k of the read packets when the $n \geq k$-th packet is read. *pg. 430*

20.7 Online sampling

The set $\mathcal{Z}_n = \{0, 1, 2, \ldots, n-1\}$ has exactly $\binom{n}{k}$ subsets of size k. We seek to design an algorithm that returns any one of these subsets with equal probability.

Problem 20.7: Design an algorithm that computes an array of size k consisting of distinct integers in the set $\{0, 1, \ldots, n-1\}$. All subsets should be equally likely and, in addition, all permutations of elements of the array should be equally likely. Your time should be $O(k)$. Your algorithm should use $O(k)$ space in addition to the k element array holding the result. You may assume the existence of a subroutine that returns integers in the set $\{0, 1, \ldots, n-1\}$ with uniform probability. *pg. 431*

20.8 House of Representatives majority

Suppose you want to place a bet on the outcome of the coming elections. Specifically, you are betting if the US House of Representatives will have a Democratic or a Republican majority. A polling company has computed the probability of winning for each candidate in the 435 individual elections. You are interested in just one

number—what is the probability that the Republican party is going to have a majority in the House?

Problem 20.8: Assuming elections are statistically independent and that the probability of a Republican winning Election i is p_i, how would you compute the probability of a Republican majority? *pg. 432*

20.9 Differentiating biased coins

Two coins that are identical in appearance are placed in a black cloth bag. One is biased towards heads—it comes up heads with probability 0.6. The other is biased towards tails—it comes up tails with probability 0.6. For both coins, the outcomes of successive tosses are independent.

Problem 20.9: You select a coin at random from the bag and toss it five times. It comes up heads three times. What is the probability that it was the coin that was biased towards tails? How many times do you need to toss the coin that is biased towards tails before it comes up with a majority of tails with probability greater than $\frac{99}{100}$? *pg. 434*

20.10 Balls and bins

Suppose n web servers interact with m clients such that each client picks a server uniformly at random. A key benefit of this approach is that no centralized control is needed. However, some servers may be idle while clients are waiting to be served. This is often modeled by balls and bins.

Problem 20.10: If m balls are thrown into n bins uniformly randomly and independently, what is the expected number of bins that do not have any balls? *pg. 435*

20.11 Random permutations

Problem 20.11: What is the expected number of fixed points of a uniformly random permutation $\sigma : \{0, 1, \ldots, n-1\} \mapsto \{0, 1, \ldots, n-1\}$, i.e., the expected cardinality of $\{i \mid \sigma(i) = i\}$? What is the expected length of the longest increasing sequence starting at $\sigma(0)$, i.e., if k is the first index such that $\sigma(k) < \sigma(k-1)$, what is the expected value of k? *pg. 436*

20.12 Expected number of die rolls

Problem 20.12: Gottfried repeatedly rolls an unbiased six-sided die. He stops when he has rolled all the six numbers on the die. How many rolls will it take, on an average, for Gottfried to see all the six numbers? *pg. 436*

20.13 Forming a triangle—1

Two numbers $u1$ and $u2$ are selected uniformly randomly and independently in the interval $[0, 1]$. Three line segments are formed, of lengths $\min(u1, u2)$, $\max(u1, u2) - \min(u1, u2)$, and $1 - \max(u1, u2)$.

Problem 20.13: What is the probability that these three segments can be assembled into a triangle? *pg. 437*

20.14 Forming a triangle—2

Problem 20.14: Solve Problem 20.13 when $u1$ is uniformly randomly in $[0, 1]$ and $u2$ is subsequently chosen uniformly randomly in $[u1, 1]$. Can you determine which of these two approaches is more likely to produce a triangle without computing the exact probabilities? *pg. 437*

20.15 Selecting the best secretary

Suppose you need to choose a secretary from a pool of n secretaries who you interview in a random order. Given any two secretaries, you can tell who is better, and the "is better" relationship is transitive. Once you interview a secretary s, you have to select or reject s right away. If you select s, the selection process stops.

Problem 20.15: Design a strategy that selects the best secretary with a probability greater than 0.25, regardless of n. *pg. 438*

20.16 The once-or-twice game

The Once-or-Twice game is played against a dealer. You pay \$1 to play the game. The dealer gets a random card from a full deck. You are shown a randomly selected card from another full deck. You have the choice of taking the card or exchanging it for another card which is also randomly selected from another full deck. You win the game if and only if the face value of your card is greater than that of dealer. If you win, you get w dollars. (The face value of an Ace is 1; the face values of Jack, Queen, and King are 11, 12, and 13, respectively.)

Problem 20.16: What is the value of w such that Once-or-Twice is a fair game, i.e., for a rational player, the expected gain is 0? *pg. 438*

20.17 The multibet card color game

In the multibet card color game, a deck of 52 playing cards is shuffled. and placed face-down on a table You can bet on the color of the top card at even odds. After you have placed your bet, the top card is revealed to you and discarded. Betting continues till the deck is exhausted. You begin with \$1. If you can bet arbitrary fractions of your bankroll, there is a simple strategy which guarantees you will win, regardless of the order in which the cards appear, $\$2^{52}/\binom{52}{26} \approx 9.08132955$. This is the maximum amount that you can guarantee that you will win. (A proof of this is

sketched at the end of the solution.) In this problem we want to find the maximum amount you can guarantee that you will win when you can bet in penny increments.

Problem 20.17: Suppose you are playing the multibet card color game and are restricted to bet in penny increments. Compute a tight lower bound on the amount that you can guarantee to win under this restriction. *pg. 439*

20.18 The one red card game

In the one red card game, a deck of 52 playing cards is shuffled and placed face-down on a table. To win, you must select a red card. You can either examine or select the top card. If you choose examine, the top card is revealed and discarded. If you choose select, the game ends—you win if it is a red card and lose otherwise. After examining 51 cards, you must select the last card.

Problem 20.18: Design a strategy that maximizes the probability of winning at the one red card game. *pg. 441*

20.19 Optimum bidding

Problem 20.19: Consider an auction for an item in which the reserve price is a random variable X uniformly distributed in $[0, 400]$. You can bid B. If your bid is greater than or equal to the reserve price, you win the auction and have to pay B. You can then sell the item for an 80% markup over the reserve price. How much should you offer for the item? *pg. 442*

20.20 The Kelly criterion

An roulette wheel has 37 pockets, numbered from 0 to 36 into which a ball is dropped. Pocket 0 is colored green; the remaining pockets are colored red if the number is odd and black if the number is even. You can bet on the ball falling into a red pocket at 50-50 odds, i.e., you get back double the amount you bet if the ball lands on a red pocket and you lose otherwise. Ordinarily, the ball lands in a pocket uniformly at random. Therefore, the probability of a bet on red paying off is $\frac{18}{37}$.

Problem 20.20: Your friend at the Acme Casino has rigged their roulette wheel to make the probability of the ball landing on red $\frac{19}{37}$. You can bet on the same color exactly 100 times; after that the casino management will be alerted. You start with \$1. On each round, you can bet any amount from 0 to your entire bankroll. What should your strategy be? *pg. 442*

20.21 The complexity of AND-OR expressions

Suppose we are to evaluate an expression of the form $((A \wedge B) \vee (C \wedge D))$, where $\wedge$ and $\vee$ are Boolean AND and OR respectively and A, B, C, and D are Boolean variables. It is natural to use *lazy evaluation*, i.e., when evaluating $A \wedge B$, if we begin with A, and it evaluates to `false`, we skip B.

We now define a restricted set of expressions. The L_0 expressions are just Boolean variables. An L_{k+1} expression is of the form $((\phi_0 \wedge \phi_1) \vee (\psi_0 \wedge \psi_1))$, where ϕ_0, ϕ_1, ψ_0, and ψ_1 are L_k expressions. All Boolean variables appearing in an L_k expression are distinct.

We want to design an algorithm for evaluating L_k expressions and want to minimize the number of variables that it reads. We do not care how much time the algorithm spends traversing the expression.

Problem 20.21: Prove that an algorithm in which the choice of the next variable to read in an L_K expression is a deterministic function of the values read up to that point must, in the worst case, read all variables to evaluate the expression. Design a randomized algorithm that reads fewer variables on an average, independent of the values assigned to the variables. *pg. 443*

Option pricing

A call option gives the owner the right to buy something—a share, a barrel of oil, an ounce of gold—at a predetermined price at a predetermined time (the "expiration date"') in the future. If the option is not priced fairly, an arbitrageur can either buy or sell the option in conjunction with other transactions and come up with a scheme of making money in a guaranteed fashion. A fair price for an option would be a price such that no arbitrage scheme can be designed around it. Problems 20.22 to 20.24 on Pages 161–162 are related to determining the fair price for an option on a stock, given the distribution of the stock price for a period of time.

20.22 Option pricing—discrete case

In the following problem, you begin with a portfolio of x_s shares and x_o options. Since you are allowed to short shares and sell options x_s and x_o may be negative. An arbitrage is a portfolio which has a negative initial value and, regardless of the movement in the share price, has a positive future value.

Consider an option to buy a stock S that currently trades at \$100. The option is to buy the stock at \$100 in 100 days. Suppose we know only two outcomes are possible—S will go to \$120 or to \$70.

If the option is priced at \$26, we have an arbitrage: buy one share for \$100 and sell four options—the initial outlay on the portfolio is $100 + 4 \times -26 = -\$4$. If the stock goes up, the portfolio is worth $120 + 4 \times -20 = \$40$. If the stock goes down, the portfolio is worth \$70. In either case, we make money with no initial investment, i.e., the option price allows for an arbitrage.

Problem 20.22: For what option price is there no opportunity for arbitrage? *pg. 444*

20.23 Option pricing with interest

Problem 20.23: Consider the same problem as Problem 20.22, with the existence of a third asset class, namely a bond. A \$1 bond pays \$1.02 in 100 days. You can borrow

money at this rate or lend it at this rate. Show there is a unique arbitrage-free price for the option and compute this price. *pg. 445*

20.24 Option pricing—continuous case

Problem 20.24: Suppose the price of Jingle stock 100 days in the future is a normal random variable with mean \$300 and standard deviation \$20. What would be the fair price of an option to buy a single share of Jingle at \$300 in 100 days? (Ignore the effect of interest rates.) *pg. 445*

CHAPTER

21

Discrete Mathematics

There is required, finally, the ratio between the fluxion of any quantity x you will and the fluxion of its power x^n. Let x flow till it becomes $x + o$ and resolve the power $(x + o)^n$ into the infinite series $x^n + nox^{n-1} + \frac{1}{2}(n^2 - n)o^2x^{n-2} + \frac{1}{6}(n^3 - 3n^2 + 2n)o^3x^{n-3} \ldots$

— *"On the Quadrature of Curves,"*
I. NEWTON, 1693

Discrete mathematics is used in algorithm design in a variety of places. Examples include combinatorial optimization, complexity analysis, and bounding probabilities. Discrete mathematics is also the source of enjoyable puzzles and challenging interview questions. Solutions can range from simple applications of the pigeon-hole principle to complex inductive reasoning.

Some of the problems in this chapter fall into the category of brain teasers where all that is needed an *aha* moment. These problems have fallen out of fashion because it is difficult to judge a candidate's ability based on whether he is able to make a tricky observation in a short period of time. However they are asked often enough that it is important to understand basic principles.

21.1 500 DOORS

Five hundred closed doors along a corridor are numbered from 1 to 500. A person walks through the corridor and opens each door. Another person walks through the corridor and closes every alternate door. Continuing in this manner, the i-th person comes and toggles the position of every i-th door starting from door i.

Problem 21.1: Which of the 500 doors are open after the 500-th person has walked through? ***pg. 446***

21.2 EFFICIENT TRIALS

You are the coach of a cycling team with 25 members and need to determine the fastest, second-fastest, and third-fastest cyclists for selection to the Olympic team.

You will be evaluating the cyclists using a time-trial course on which only five cyclists can race at a time. You can use the completion times from a time-trial to rank the five cyclists amongst themselves—no ties are possible. Because conditions can change over time, you cannot compare performances across different time-trials. The

relative speeds of cyclists does not change—if a beats b in one time-trial and b beats c in another time-trial, then a is guaranteed to beat c if they are in the same time-trial.

Problem 21.2: What is the minimum number of five man time-trials needed to determine the top three cyclists from a set of 25 cyclists? *pg. 447*

21.3 Space-time intersections

Albert starts climbing a mountain at 9:00 a.m. on Saturday. He reaches the summit at 5:00 p.m. He camps at the summit overnight and descends the mountain on Sunday. He begins and ends at the same time and follows exactly the same route. His speeds may vary and he may take breaks at different places.

Problem 21.3: Prove that there exists a place such that Albert is at that place at the same time on Sunday as he was on Saturday. *pg. 447*

21.4 Hershey bar

A Hershey bar is modeled as a rectangle of $m \times n$ rectangle-shaped pieces. You can take a bar and break it along a horizontal or vertical axis into two bars.

Problem 21.4: How would you break a 4×4 bar into 16 pieces using as few breaks as possible? *pg. 447*

21.5 Picking up coins, don't lose

Sixteen coins are placed in a line, as in Figure 4.6 on Page 44. Two players, F and S, take turns at choosing one coin each—they can only choose from the two coins at the ends of the line. Player F goes first. The game ends when all the coins have been picked up. The player whose coins have the higher total value wins. Each player must select a coin when it is his turn, guaranteeing that the game ends in sixteen turns.

Problem 21.5: If you want to ensure you do not lose, would you rather be F or S? *pg. 447*

21.6 $n \times n$ chomp

The game called $n \times n$ chomp consists of an $n \times n$ rectangle in the upper right quadrant in the Cartesian plane, with the lower leftmost point at $(0, 0)$. The block $(0, 0)$ is poisoned. Two players take turns at taking a bite out of the rectangle. A bite removes a square and all squares above and to the right. The first player to eat the poisoned square loses.

Problem 21.6: Assuming the players have infinite computational resources at their disposal, who will win $n \times n$ chomp? *pg. 448*

21.7 $n \times 2$ CHOMP

Problem 21.7: Solve Problem 21.6 on the facing page if the rectangle is n long along the x-axis, and two long along the y-axis. *pg. 449*

21.8 $n \times m$ CHOMP

Problem 21.8: Solve Problem 21.6 on the preceding page if the rectangle is of dimension $n \times m$. *pg. 449*

21.9 MAILBOX PLACEMENT

A total of n apartment buildings are coming up on a new street. The postal service wants to place a single mailbox on the street. Their objective is to minimize the total distance that residents have to walk to collect their mail each day.

Problem 21.9: Building i has r_i residents, and is at distance d_i from the beginning of the street. Devise an algorithm that computes a distance m from the beginning of the street for the mailbox that minimizes the total distance, that residents travel to get to the mailbox, i.e., minimizes $\sum_{i=0}^{n-1} r_i|d_i - m|$. *pg. 449*

21.10 THE GASUP PROBLEM

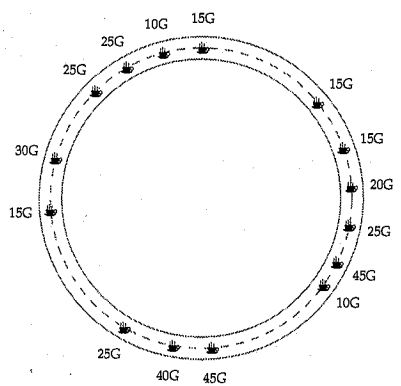


Figure 21.1: The length of the circular route is 7200 miles, and your vehicle gets 20 miles per gallon. The distance between successive gas stations is proportional to the angle they subtend at the center.

In the gasup problem, n cities are arranged on a circular road. You need to visit all the n cities and come back to the starting city. A certain amount of gas is available at each city. The total amount of gas is equal to the amount of gas required to go around the road once. Your gas tank has unlimited capacity. Call a city ample if you can begin at it with an empty tank, travel through all the remaining cities, refilling at each, and return to it. An instance of this problem is given in Figure 21.1.

Problem 21.10: Given an instance of the gasup problem, how would you efficiently compute an ample city if one exists? *pg. 450*

21.11 Closest palindrome

A palindromic string is one which reads the same when it is reversed. For example, the string "malayalam" is a palindrome. An integer can be represented as a string of digits, so we can speak of palindromic integers.

Problem 21.11: Write a function that takes a nonnegative integer x and returns, as a string, the integer closest to x whose decimal representation is a palindrome. For example, given 1224, you should return 1221. *pg. 451*

21.12 The maximum product of $(n-1)$ numbers

You are given an array A of n elements, $n \geq 2$, and are asked to find the $n-1$ elements in A which have the largest product.

One approach is to form the product $P = \prod_{i=0}^{n-1} A[i]$, and then find the maximum of the n terms $P_i = P/A[i]$; this takes $n-1$ multiplications and n divisions. Suppose because of finite precision considerations we cannot use the division-based approach described above; we can only use multiplications. The brute-force solution entails computing all $\binom{n}{n-1} = n$ products of $n-1$ elements; each such product takes $n-2$ multiplications.

Problem 21.12: Given an array A with n elements, compute $\max_{j=0}^{n-1} \frac{\prod_{i=0}^{n-1} A[i]}{A[j]}$ in $O(n)$ time without using division. Can you design an algorithm that runs in $O(1)$ space and $O(n)$ time? Array entries may be positive, negative, or 0. *pg. 452*

21.13 Height determination

You need to test the design of a protective case. Specifically, the case can protect the enclosed device from a fall from up to some number of floors, and you want to determine what that number of floors is. You want to achieve this using no more than c cases. An additional constraint is that you can perform only d drops before the building supervisor stops you.

You know that there exists a floor x such that the case will break if it is dropped from any floor x or higher but will remain intact if dropped from a floor below x. The ground floor is numbered zero, and it is given that the case will not break if dropped from the ground floor.

Problem 21.13: Given c cases and d drops, what is the maximum number of floors that you can test in the worst-case? *pg. 454*

21.14 Symmetric-Whack-a-Mole

In the game of Symmetric-Whack-a-Mole, moles are in one of two states—*up* or *down*. The player has a hammer which he can use to "whack" a mole on the head, and thereby flip its state.

Problem 21.14: Moles are numbered from 0 to $n-1$. Mole m has a set of neighboring moles. Whacking m when it is up results in it and all of its neighbors flipping state. Given a set of moles, the neighbors for each mole, and an initial assignment of up/down states for each mole, compute a sequence of whacks (if one exists) that results in each mole being in the down state. *pg. 455*

21.15 Celebrity identification (☺)

For any two distinct people a and b, a may or may not know b. However, the "knows" relation is not symmetric, which means that a may know b, but b may not know a. The knows relation is anti-reflexive, i.e., a does not know a. At a party, everyone knows someone else. Now a celebrity joins the party—everyone knows him, but he knows no one.

Problem 21.15: Let F be an $n \times n$ Boolean 2D array representing the "knows" relation for n people; $F[a][b]$ is `true` iff a knows b, and $F[a][a]$ is always `false`. Design an $O(n)$ algorithm to find the celebrity. *pg. 456*

21.16 Ramsey theory (☺)

In 1930, Frank Ramsey wrote a paper titled *"On a problem in formal logic"* which initiated an entirely new field of discrete mathematics called "Ramsey Theory" in his honor. He proved what is now called Ramsey's theorem as an intermediate lemma in a bigger proof. The problem below illustrates Ramsey's theorem.

Problem 21.16: Six guests attend a party. Any two guests either know each other or do not know each other. Prove that there exists a subset of three guests who either all know each other or all do not know each other. *pg. 457*

21.17 Tournaments and Hamiltonian paths

A tournament is a directed graph obtained by assigning a direction for each edge in an undirected complete graph in which every pair of distinct vertices is connected by an edge.

Problem 21.17: Prove that every tournament has a Hamiltonian path, i.e., a path that includes each vertex exactly once. *pg. 457*

21.18 Stable assignment (☺)

Consider a department with n new graduate students and n professors. Each student has ordered all the professors based on how keen he is to work with them. Similarly, each professor has an ordered list of all the students.

Problem 21.18: Design an algorithm which takes the preference lists of the students and the professors and pairs students one-to-one with professors subject to the constraint that there do not exist student-professor pairings $(s0, p0)$ and $(s1, p1)$ such that $s0$ prefers $p1$ to $p0$ and $p1$ prefers $s0$ to $s1$. (The preferences of $p0$ and $s1$ are not important.) *pg. 459*

21.19 Dancing with the stars

You are organizing a celebrity dance for charity. Specifically, a number of celebrities have offered to be partners for a ballroom dance. The general public has been invited to offer bids on how much they are willing to pay for a dance with each celebrity.

Problem 21.19: Design an algorithm for pairing bidders with celebrities to maximize the revenue from the dance. Each celebrity cannot dance more than once, and each bidder cannot dance more than once. Assume that the set of celebrities is disjoint from the set of bidders. How would you modify your approach if all bids were for the same amount? What if celebrities and bidders are not disjoint? *pg. 460*

21.20 Tiling a chessboard

You are given a chessboard and 31 dominoes. The dominoes are rectangles, and each domino is equal to two squares from the chessboard.

If we remove two diagonally opposite squares from the chessboard, we cannot cover the chessboard with the 31 dominoes, since each domino will cover one white and one black square, and the two removed squares must be of the same color.

In this problem we consider the converse.

Problem 21.20: Suppose two squares of opposite colors are removed from a chessboard. Design an algorithm for finding a way to cover the remaining squares using 31 dominoes, if a covering exists. *pg. 460*

21.21 Team photo day—3

Problem 21.21: This problem is a continuation of Problems 13.6 on Page 100 and 16.7 on Page 135. Design an efficient algorithm for computing the minimum number of subsets of teams so that (1.) the teams in each subset can be organized to appear in a single photograph without violating the placement constraint, and (2.) each team appears in exactly one subset. *pg. 462*

21.22 Largest common rooted subtree

Define rooted trees A and B to be isomorphic if

- both are `null`, or
- they have the same number of children, and there exists a one-to-one function f from the children of A to the children of B such that for all u, $f(u)$ is isomorphic to u.

Define a subtree of a rooted tree A to be a subset S of the nodes of A that form a rooted tree when the parent-child relationship from A is applied to S.

Figure 21.2 on the next page shows two rooted trees, $T1$ and $T2$. The gray nodes indicate a subtree of $T1$ that is isomorphic to a subtree of $T2$. No larger subtree of $T1$ is isomorphic to a subtree of $T2$.

Problem 21.22: Let A and B be rooted trees. Design a polynomial time algorithm for computing a largest common rooted subtree of A and B. *pg. 463*

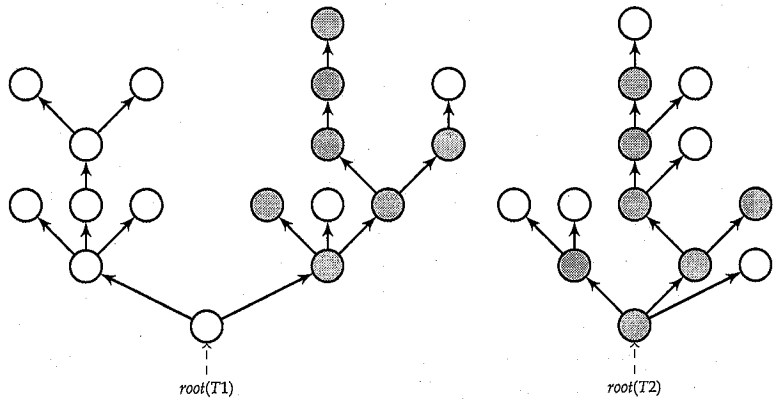


Figure 21.2: Largest common rooted subtree.

21.23 Team elimination (☹)

Towards the end of a season, sportswriters describe Team a as being mathematically eliminated if, no matter what the outcomes of the remaining games are, some other team will end up with more wins than a.

Problem 21.23: Consider a league in which teams are numbered from 0 to $n-1$. At a certain point in the season, Team i has won W_i games, and has $R_{i,j}$ games remaining with Team j. Each game will end in a win for one team and a loss for the other team. Show how the problem of determining whether Team a is mathematically eliminated can be solved using maximum flow. *pg. 464*

21.24 Rounding a matrix (☹)

Let A be an $m \times n$ matrix of nonnegative real numbers. Define a *rounding* of A to be an $m \times n$matrix F_A such that for all i and j, $F_A[i,j] = \lfloor A[i,j] \rfloor$ or $\lceil A[i,j] \rceil$, for all i, $\sum_{j=0}^{n-1} F_A[i,j] = \sum_{j=0}^{n-1} A[i,j]$, and for all j, $\sum_{i=0}^{m-1} F_A[i,j] = \sum_{i=0}^{m-1} A[i,j]$.

For example, if

$$A = \begin{bmatrix} 1.4 & 2.1 & 3.5 \\ 4.0 & 5.8 & 6.2 \\ 7.6 & 8.1 & 9.3 \end{bmatrix}$$

then

$$F_A = \begin{bmatrix} 2 & 2 & 3 \\ 4 & 6 & 6 \\ 7 & 8 & 10 \end{bmatrix}$$

is a rounding of A.

Problem 21.24: Design an efficient algorithm for computing a rounding of amatrix, if one exists. *pg. 466*

21.25 Common knowledge

An explorer comes to the Isle of Logic, which contains 100 inhabitants, half of whom have blue eyes. The remaining inhabitants have green eyes. The green eyes are indicative of a disease that brings all the island inhabitants in danger. The inhabitants have an understanding that whenever someone learns that they have green eyes, they must leave the island; they never leave the island for any other reason. Inhabitants are too polite to inform anyone of their eye color.

The inhabitants assemble each day at exactly 9:00 a.m., they see each other, and then go back to their own houses. They never see anyone else for the rest of the day. Furthermore, they are capable of instant logical reasoning.

The explorer visits one of their daily assemblies and says, "That's interesting—some of you have blue eyes and some of you have green eyes".

Problem 21.25: What follows after the explorer visits the Isle of Logic? The explorer seems to have added no new knowledge since each inhabitant already knows that some inhabitants have blue eyes and some have green eyes. Why does his observation change the equilibrium? *pg. 466*

21.26 Computing an optimum mixed-strategy

A payoff matrix A is an $m \times n$ 2D array of real numbers. Player 1 selects Row i with probability p_i and Player 2 selects Column j with probability q_j. Player 1 receives $A[i][j]$ as his payoff, and Player 2 receives $-A[i][j]$.

Problem 21.26: Given a payoff matrix, compute values $p_0, p_1, \ldots, p_{m-1}$ for Player 1 that minimize the maximum payoff for Player 2. Assume Player 2 knows $p_0, p_1, \ldots, p_{m-1}$. *pg. 467*

Part III

Solutions

C++11

C++11 adds a number of features that make for elegant and efficient code. The C++11 constructs used in the solution code are summarized below.

- The `auto` attribute assigns the type of a variable based on the initializer expression.
- The enhanced range-based for-loop allows for easy iteration over a list of elements.
- The `emplace_front` and `emplace_back` methods add new elements to the beginning and end of the container. They are more efficient than `push_front` and `push_back`, and are variadic, i.e., takes a variable number arguments. The `emplace` method is similar and applicable to containers where there is only one way to insert (e.g., a stack or a map)
- The `array` type is similar to ordinary arrays, but supports `.size()` and boundary checking. (It does not support automatic resizing.)
- The `tuple` type implements an ordered set.
- Anonymous functions ("lambdas") can be written via the `[]` notation, as illustrated in Solution 13.2 on Page 292.
- An initializer list uses the `{}` notation to avoid having to make explicit calls to constructors when building list-like objects.
- The function `iota(ForwardIterator f, ForwardIterator l, T v)` fills the range `[f, l)` with sequentially increasing values, starting with `v` and repetitively evaluating `++v`.

C++ for Java developers

C++ is an order of magnitude more complex than Java. Here are some facts about C++ that can help Java programmers better understand the solution code.

- Operators in C++ can be overloaded. For example, < can be applied to comparing `BigNumber` objects. The array indexing operator (`[]`) is often overloaded for unordered maps and tree maps, e.g., `map[k]` returns the value associated with key `k`.
- Java's `HashMap` and `HashSet` correspond to C++'s `unordered_map` and `unordered_set`, respectively. Java's `TreeSet` and `TreeMap` correspond to C++'s `set` and `map`.
- For `set`, the comparator is the second argument to the template specification. For `map`, the comparator is the third argument to the template specification. (If < is overloaded, the comparator is optional in both cases.)
- For `unordered_map` the first argument is the key type, the second is the value type, and the third (optional) is the hash function. For `unordered_set` the first argument is the key type, the second (optional) is the hash function, the third (optional) is the equals function. The class may simply overload ==, i.e., implement the method `operator==`. See Solution 12.10 on Page 282 for an example.
- C++ uses streams for input-output. The overloaded operators « and » are used to read and write primitive types and objects from and to streams.

- The `::` notation is used to invoke a static member function or refer to a static field.
- C++ has a built-in `pair` class used to represent arbitrary pairs.
- A `static_cast` is used to cast primitive types, e.g., `int` to `double`, as well as an object to a derived class. The latter is not checked at run time. The compiler checks obvious incompatibilities at compile time.
- A `shared_pointer` is a pointer with a reference count which the runtime system uses to implement automatic garbage collection.

Problem 5.1, *pg. 47*: *How would you go about computing the parity of a very large number of 64-bit nonnegative integers?*

Solution 5.1: The fastest algorithm for manipulating bits can vary based on the underlying hardware.

The time taken to directly compute the parity of a single number is proportional to the number of bits:

```
short parity1(unsigned long x) {
  short result = 0;
  while (x) {
    result ^= (x & 1);
    x >>= 1;
  }
  return result;
}
```

A neat trick that erases the least significant bit of a number in a single operation can be used to improve performance in the best and average cases:

```
short parity2(unsigned long x) {
  short result = 0;
  while (x) {
    result ^= 1;
    x &= (x - 1);  // drops the LSB of x
  }
  return result;
}
```

However, when you have to perform a large number of parity operations, and more generally, any kind of bit fiddling operation, the best way to proceed is to precompute the answer and store it in an array. Depending upon how much memory is at your disposal, and how much fits efficiently in cache, you can vary the size of the lookup table. Below is an example implementation where you build a lookup table "`precomputed_parity`" that stores the parity of any 16-bit number i as `precomputed_parity[i]`. This array can either be constructed during static initialization or dynamically—a flag bit can be used to indicate if the entry at a location is uninitialized. Once you have this array, you can implement the parity function as follows:

```
short parity3(const unsigned long &x) {
  return precomputed_parity[x >> 48] ^
```

```
        precomputed_parity[(x >> 32) & 0xFFFF] ^
        precomputed_parity[(x >> 16) & 0xFFFF] ^
        precomputed_parity[x & 0xFFFF];
}
```

We are assuming that the `short` type is 16 bits, and the `unsigned long` is 64 bits. The operation `x » 48` returns the value of `x` right-shifted by 48 bits. Since `x` is unsigned, the C++ language standard guarantees that bits vacated by the shift operation are zero-filled. (The result of a right-shift for signed quantities, is implementation dependent, e.g., either 0 or the sign bit may be propagated into the vacated bit positions.)

Problem 5.2, *pg. 47*: *A 64-bit integer can be viewed as an array of 64 bits, with the bit at index 0 corresponding to the least significant bit, and the bit at index 63 corresponding to the most significant bit. Implement code that takes as input a 64-bit integer x and swaps the bits at indices i and j.*

Solution 5.2: First determine if the bits at indices i and j differ. The bit at index i is identified by right shifting by i and ANDing with 1; the bit at index j is handled similarly. Generally speaking, a right-shift of an integer may be signed (the most significant bit is replicated) or unsigned (0s are inserted). However, it makes no difference to our application, since the bits shifted in are ANDed with 0s. Here is the code in C++:

```
long swap_bits(long x, const int &i, const int &j) {
  if (((x >> i) & 1) != ((x >> j) & 1)) {
    x ^= (1L << i) | (1L << j);
  }
  return x;
}
```

Problem 5.3, *pg. 48*: *Write a function that takes a 64-bit integer x and returns a 64-bit integer consisting of the bits of x in reverse order.*

Solution 5.3: Similar to computing parity (Problem 5.1 on Page 47), the fastest way to reverse bits is to build a precomputed array `precomputed_reverse` such that for every 16-bit number i, `precomputed_reverse[i]` holds the bit-reversed i:

```
long reverse_bits(const long &x) {
  return precomputed_reverse[(x >> 48) & 0xFFFF] |
         precomputed_reverse[(x >> 32) & 0xFFFF] << 16 |
         precomputed_reverse[(x >> 16) & 0xFFFF] << 32 |
         precomputed_reverse[x & 0xFFFF] << 48;
}
```

Problem 5.4, *pg. 48*: *Suppose $x \in S_k$, and k is not 0 or 64. How would you compute $y \in S_k \setminus \{x\}$ such that $|y - x|$ is minimum?*

Solution 5.4: The number y can be computed by iterating through the bits of x, from the least significant bit to the most significant bit and swapping the first two consecutive bits that differ. Intuitively, this works because we want to change the least significant bits possible. (Note that simply swapping the least significant bit with next least significant bit that differs from the least significant bit does not work, e.g., 1011100 provides a counterexample.)

```
unsigned long closest_int_same_bits(unsigned long x) {
  for (int i = 0; i < 63; ++i) {
    if (((x >> i) & 1) ^ ((x >> (i + 1)) & 1)) {
      x ^= (1UL << i) | (1UL << (i + 1));  // swaps bit-i and bit-(i + 1)
      return x;
    }
  }

  // Throw error if all bits of x are 0 or 1
  throw invalid_argument("all bits are 0 or 1");
}
```

Problem 5.5, *pg. 48*: *Implement a method that takes as input a set S of distinct elements, and prints the power set of S. Print the subsets one per line, with elements separated by commas.*

Solution 5.5: The key to solving this problem is realizing that for a given ordering of the elements of S, there exists a one-to-one correspondence between the $2^{|S|}$ bit arrays of length $|S|$ and the set of all subsets of S—the 1s in the $|S|$-length bit array v indicate the elements of S in the subset corresponding to v.

For example, if $S = \{g, l, e\}$ and the elements are ordered $g < l < e$, the bit array $\langle 0, 1, 1 \rangle$ denotes the subset $\{l, e\}$.

If $|S|$ is less than or equal to the number of bits used to represent an integer on the architecture (or language) we are working on, we can enumerate bit arrays by enumerating integers in $[0, 2^{|S|} - 1]$ and examining the indices of bits set in these integers. These indices are determined by first isolating the most significant bit by computing $y = x \& !(x - 1)$ and then getting the index by computing $\lg y$.

```
template <typename T>
void generate_power_set(const vector<T> &S) {
  for (int i = 0; i < (1 << S.size()); ++i) {
    int x = i;
    while (x) {
      int tar = log2(x & ~(x - 1));
      cout << S[tar];
      if (x &= x - 1) {
        cout << ',';
      }
    }
    cout << endl;
  }
}
```

In practice, it would likely be faster to iterate through all the bits in x, one at a time.

Alternately, we can use recursion. We make one call with the i-th element and one call without the i-th element. The time complexity is $O(|S|2^{|S|})$. The space complexity is $O(|S|)$ which comes from the maximum stack depth as well as the maximum size of a subset.

```
template <typename T>
void generate_power_set_helper(const vector<T> &S, int idx, vector<T> &res) {
  if (res.empty() == false) {
    // Print the subset
    copy(res.cbegin(), res.cend() - 1, ostream_iterator<T>(cout, ","));
    cout << res.back();
  }
  cout << endl;

  for (int i = idx; i < S.size(); ++i) {
    res.emplace_back(S[i]);
    generate_power_set_helper(S, i + 1, res);
    res.pop_back();
  }
}

template <typename T>
void generate_power_set(const vector<T> &S) {
  vector<T> res;
  generate_power_set_helper(S, 0, res);
}
```

Variant 5.5.1: Print all subsets of size k of $\{1, 2, 3, \ldots, n\}$.

Problem 5.6, *pg. 49*: *Implement string/integer inter-conversion functions. Use the following function signatures:* `String intToString(int x)` *and* `int stringToInt(String s)`.

Solution 5.6: For a positive integer x, we iteratively divide x by 10, and record the remainder till we get to 0. This yields the result from the least significant digit, and needs to be reversed. If x is negative, we record that, and negate x, adding a '-' afterward. If x is 0, our code breaks out of the iteration without writing any digits, in which case we need to explicitly set a 0. In C++ code:

```
string intToString(int x) {
  bool is_negative;
  if (x < 0) {
    x = -x, is_negative = true;
  } else {
    is_negative = false;
  }

  string s;
  while (x) {
    s.push_back('0' + x % 10);
```

```
    x /= 10;
  }
  if (s.empty()) {
    return {"0"};  // x is 0
  }

  if (is_negative) {
    s.push_back('-');
  }
  reverse(s.begin(), s.end());
  return s;
}

int stringToInt(const string &s) {
  bool is_negative = s[0] == '-';
  int x = 0;

  for (int i = is_negative; i < s.size(); ++i) {
    if (isdigit(s[i])) {
      x = x * 10 + s[i] - '0';
    } else {
      throw invalid_argument("illegal input");
    }
  }
  return is_negative ? -x : x;
}
```

Problem 5.7, *pg. 49*: *Write a function that performs base conversion. Specifically, the input is an integer base b_1, a string s, representing an integer x in base b_1, and another integer base b_2; the output is the string representing the integer x in base b_2. Assume $2 \leq b_1, b_2 \leq 16$. Use "A" to represent 10, "B" for 11, ..., and "F" for 15.*

Solution 5.7: We can use a reductionist approach to solve this problem. We have seen how to convert integers to strings in Solution 5.6 on the preceding page; this approach works for any base. Converting from strings is the reverse of this process. We can therefore convert base b_1 string s to a base 10 integer x, and then convert x to a base b_2 string ans. Following is the code in C++:

```
string convert_base(const string &s, const int &b1, const int &b2) {
  bool neg = s.front() == '-';
  int x = 0;
  for (int i = (neg == true ? 1 : 0); i < s.size(); ++i) {
    x *= b1;
    x += isdigit(s[i]) ? s[i] - '0' : s[i] - 'A' + 10;
  }

  string ans;
  while (x) {
    int r = x % b2;
    ans.push_back(r >= 10 ? 'A' + r - 10: '0' + r);
    x /= b2;
  }

```

```
  if (ans.empty()) {
    ans.push_back('0');
  }
  if (neg) {
    ans.push_back('-');
  }
  reverse(ans.begin(), ans.end());
  return ans;
}
```

Problem 5.8, *pg. 49*: *Write a function that converts Excel column ids to the corresponding integer, with "A" corresponding to 1. The function signature is* `int ssDecodeColID(string)`*; you may ignore error conditions, such as* `col` *containing characters outside of* $[A, Z]$*. How would you test your code?*

Solution 5.8: This problem is similar to the problem of converting a string representing a base-26 number to the corresponding integer, except that "A" corresponds to 1 not 0.

```
int ssDecodeColID(const string &col) {
  int ret = 0;
  for (const char &c : col) {
    ret = ret * 26 + c - 'A' + 1;
  }
  return ret;
}
```

Good test cases are around boundaries, e.g., "A", "B", "Y", "Z", "AA", "AB", "ZY", "ZZ", and some random strings, e.g., "M", "BZ", "CCC".

Problem 5.9, *pg. 50*: *Let A be an array of n integers. Write an encode function that returns a string representing the concatenation of the Elias gamma codes for $\langle A[0], A[1], \ldots, A[n-1] \rangle$ in that order, and a decode function that takes a string s assumed to be generated by the encode function, and returns the array that was passed to the encode function.*

Solution 5.9: The code follows in a straightforward way from the above specifications. Each encoded number starts with one fewer 0s than bits in the number, which allows us to uniquely determine the length of the result.

```
string trans_int_to_binary(int decimal) {
  string ret;
  while (decimal) {
    ret.insert(0, 1, '0' + (decimal & 1));
    decimal >>= 1;
  }
  return ret;
}

string encode(const vector<int> &A) {
  string ret = "";
  for (const int &a : A) {
```

```
    string binary = trans_int_to_binary(a);
    binary.insert(0, binary.size() - 1, '0');  // prepend 0s
    ret += binary;
  }
  return ret;
}

int trans_binary_to_int(const string &binary) {
  int ret = 0;
  for (const char &c : binary) {
    ret = (ret << 1) + c - '0';
  }
  return ret;
}

vector<int> decode(const string &s) {
  vector<int> ret;
  int idx = 0;
  while (idx < s.size()) {
    // Count the number of consecutive 0s
    int zero_idx = idx;
    while (zero_idx < s.size() && s[zero_idx] == '0') {
      ++zero_idx;
    }

    int len = zero_idx - idx + 1;
    ret.emplace_back(trans_binary_to_int(s.substr(zero_idx, len)));
    idx = zero_idx + len;
  }
  return ret;
}
```

Problem 5.10, *pg. 50*: *Design an algorithm for computing the GCD of two numbers without using multiplication, division or the modulus operators.*

Solution 5.10: The idea is to use recursion, the base case being where one of the arguments is 0. Otherwise, we check if none, one or both numbers are even. If both are even, we compute the GCD of these numbers divided by 2, and return that result times 2; if one is even, we half it, and return the GCD of the resulting pair; if both are odd, we subtract the smaller from the larger and return the GCD of the resulting pair. Multiplication by 2 is trivially implemented with a single left shift. Division by 2 is done with a single right shift.

Note that the last step leads to a recursive call with one even and one odd number. Consequently, in every two calls, we reduce the combined bit length of the two numbers by at least one, meaning that the run time complexity is proportional to the sum of the lengths of the arguments.

Languages such as Java and Python include libraries for manipulating integers of arbitrary length, making them ideally suited for our application.

```
private static BigInteger TWO = new BigInteger("2");

```

```
private static boolean isOdd(BigInteger x) {
  return x.testBit(0);
}

private static boolean isEven(BigInteger x) {
  return !isOdd(x);
}

public static BigInteger GCD(BigInteger x, BigInteger y) {
  if (x.equals(BigInteger.ZERO)) {
    return y;
  } else if (y.equals(BigInteger.ZERO)) {
    return x;
  } else if (isEven(x) && isEven(y)) {
    x = x.shiftRight(1);
    y = y.shiftRight(1);
    return TWO.multiply(GCD(x, y));
  } else if (isOdd(x) && isEven(y)) {
    return GCD(x, y.shiftRight(1));
  } else if (isOdd(y) && isEven(x)) {
    return GCD(y, x.shiftRight(1));
  } else if (x.compareTo(y) <= 0) {
    return GCD(x, y.subtract(x));
  } else {
    return GCD(y, x.subtract(y));
  }
}
```

Problem 5.11, *pg. 50*: *Write a function that takes a single positive integer argument n ($n \geq 2$) and return all the primes between 1 and n.*

Solution 5.11: We use a bit-vector `is_prime` of length $n+1$ to encode the set of primes. Initialize each entry to 1. The entry `is_prime[i]` will eventually be set to 0 iff i is not a prime. Set p to 2. Count in increments of p and "mark" (set the corresponding entry in `is_prime` to 0) each number greater than p in the count to be a non-prime (since it is divisible by p). Update p to the next unmarked number, and iterate.

This approach can be improved somewhat by ignoring even numbers, and not allocating entries for i less than 3. The count can also start from p^2 instead of p, since all numbers kp, where $k < p$ have already been marked. The code below reflects these optimizations.

```
// Given n, return the primes from 1 to n
vector<int> generate_primes_from_1_to_n(const int &n) {
  int size = floor(0.5 * (n - 3)) + 1;
  // is_prime[i] represents (2i + 3) is prime or not
  vector<int> primes;  // stores the primes from 1 to n
  primes.emplace_back(2);
  vector<bool> is_prime(size, true);
  for (long i = 0; i < size; ++i) {
    if (is_prime[i]) {
      int p = (i << 1) + 3;
      primes.emplace_back(p);
```

```
      // Sieving from p^2, whose index is 2i^2 + 6i + 3
      for (long j = ((i * i) << 1) + 6 * i + 3; j < size; j += p) {
        is_prime[j] = false;
      }
    }
  }
  return primes;
}
```

Problem 5.12, *pg. 50*: *Let R and S be xy-aligned rectangles in the Cartesian plane. Write a function which tests if R and S have a nonempty intersection. If the intersection is nonempty, return the rectangle formed by their intersection.*

Solution 5.12: Let the given rectangles be $R = ((R_x, R_y), R_w, R_h)$ and $S = ((S_x, S_y), S_w, S_h)$. Observe that the rectangles definitely do not intersect if $I_x = [R_x, R_x + R_w] \cap [S_x, S_x + S_w] = \emptyset$; similarly, they definitely do not intersect if $I_y = [R_y, R_y + R_h] \cap [S_y, S_y + S_h] = \emptyset$.

Conversely, any point $p = (x, y)$ such that $x \in I_x$ and $y \in I_y$ lies in both R and S. Suppose $I_x = [a_x, b_x]$ and $I_y = [a_y, b_y]$; then the desired rectangle is $((a_x, a_y), b_x - a_x, b_y - a_y)$.

```
class Rectangle {
  public:
    int x, y, width, height;
};

bool is_intersect(const Rectangle &R, const Rectangle &S) {
  return R.x <= S.x + S.width && R.x + R.width >= S.x &&
         R.y <= S.y + S.height && R.y + R.height >= S.y;
}

Rectangle intersect_rectangle(const Rectangle &R, const Rectangle &S) {
  if (is_intersect(R, S)) {
    return {max(R.x, S.x), max(R.y, S.y),
            min(R.x + R.width, S.x + S.width) - max(R.x, S.x),
            min(R.y + R.height, S.y + S.height) - max(R.y, S.y)};
  } else {
    return {0, 0, -1, -1};  // no intersection
  }
}
```

Variant 5.12.1: Given four points in the plane, how would you check if they are the vertices of an xy-aligned rectangle?

Variant 5.12.2: How would you check if two rectangles, not necessarily xy-aligned, intersect?

Problem 5.13, *pg. 51*: *Write a function that multiplies two unsigned positive integers. The only operators you are allowed to use are assignment and the bitwise operators, i.e., », «, |, &,*

~, ^. (In particular, you cannot use increment or decrement.) You may use loops, conditionals and functions that you write yourself; other functions are allowed.

Solution 5.13: We mimic the grade school algorithm for multiplication. Suppose we are to multiply x and y. We initialize sum to 0 and iterate through the bits of x, adding $2^k y$ to sum if bit k of x is 1.

We implement addition itself by mimicking the grade school algorithm for addition. This consists computing the sum bit-by-bit, and "rippling" the carry along. We use a bitmask that identifies the k-th bits; it also serves to tell us when all bits have been read.

```
unsigned add_no_operator(const unsigned &a, const unsigned &b) {
  unsigned sum = 0, carryin = 0, k = 1;
  while (k) {
    unsigned ak = a & k, bk = b & k;
    unsigned carryout =  (ak & bk) | (ak & carryin) | (bk & carryin);
    sum |= (ak ^ bk ^ carryin);
    carryin = carryout << 1;
    k <<= 1;
  }
  return sum;
}

unsigned multiply_no_operator(const unsigned &x, const unsigned &y) {
  unsigned sum = 0, k = 1, scaled_y = y;
  while (k) {
    // Examine the k-th bit of x
    if (x & k) {
      sum = add_no_operator(sum, scaled_y);
    }
    k <<= 1;
    scaled_y <<= 1;
  }
  return sum;
}
```

Problem 5.14, *pg. 51*: *Given two positive integers x and y, how would you compute x/y if the only operators you can use are addition, subtraction, and multiplication?*

Solution 5.14: We can use the following recursion:

$$\frac{x}{y} = \begin{cases} 0, & \text{if } x < y; \\ 1 + \frac{(x-y)}{y}, & \text{otherwise.} \end{cases}$$

This is not efficient by itself, but we can improve it by computing the largest k such that $2^k y \leq x$, in which case the recursive step is $2^k + \frac{(x-2^k y)}{y}$.

```
unsigned divide_x_y(const unsigned &x, const unsigned &y) {
  if (x < y) {
    return 0;
  }
```

```
  int power = 0;
  while ((1U << power) * y <= x) {
    ++power;
  }
  unsigned part = 1U << (power - 1);
  return part + divide_x_y(x - part * y, y);
}
```

Problem 6.1, pg. 52: *Write a function that takes an array A and an index i into A, and rearranges the elements such that all elements less than A[i] appear first, followed by elements equal to A[i], followed by elements greater than A[i]. Your algorithm should have O(1) space complexity and O(|A|) time complexity.*

Solution 6.1: This problem is conceptually straightforward: maintain four groups, *bottom* (elements less than `pivot`), *middle* (elements equal to `pivot`), *unclassified*, and *top* (elements greater than `pivot`). These groups are stored in contiguous order in A. To make this partitioning run in $O(1)$ space, we use `smaller`, `equal`, and `larger` pointers to track these groups in the following way:

- *bottom*: stored in subarray $A[0 : \texttt{smaller} - 1]$.
- *middle*: stored in subarray $A[\texttt{smaller} : \texttt{equal} - 1]$.
- *unclassified*: stored in subarray $A[\texttt{equal} : \texttt{larger}]$.
- *top*: stored in subarray $A[\texttt{larger} + 1 : |A| - 1]$.

We explore elements of *unclassified* in order, and classify the element into one of *bottom*, *middle*, and *top* groups according to the relative order between the incoming unclassified element and `pivot`. Each iteration decreases the size of *unclassified* group by 1, and the time spent within each iteration is constant, implying the time complexity is $\Theta(|A|)$.

The implementation is short but tricky, pay attention to the movements of pointers.

```
template <typename T>
void dutch_flag_partition(vector<T> &A, const int &pivot_index) {
  T pivot = A[pivot_index];
  /**
   * Keep the following invariants during partitioning:
   * bottom group: A[0 : smaller - 1]
   * middle group: A[smaller : equal - 1]
   * unclassified group: A[equal : larger]
   * top group: A[larger + 1 : A.size() - 1]
   */
  int smaller = 0, equal = 0, larger = A.size() - 1;
  // When there is any unclassified element
  while (equal <= larger) {
    // A[equal] is the incoming unclassified element
    if (A[equal] < pivot) {
      swap(A[smaller++], A[equal++]);
    } else if (A[equal] == pivot) {
      ++equal;
    } else {  // A[equal] > pivot
      swap(A[equal], A[larger--]);
```

```
    }
  }
}
```

ϵ-Variant 6.1.1: Assuming that keys take one of three values, reorder the array so that all objects of the same key appear in the same subarray. The order of the subarrays is not important. For example, both Figures 6.1(b) and 6.1(c) on Page 53 are valid answers for Figure 6.1(a) on Page 53. Use $O(1)$ additional space and $O(|A|)$ time.

ϵ-Variant 6.1.2: Given an array A of objects with keys that takes one of four values, reorder the array so that all objects that have the same key appear in the same subarray. Use $O(1)$ additional space and $O(|A|)$ time.

ϵ-Variant 6.1.3: Given an array A of objects with Boolean-valued keys, reorder the array so that all objects that have the same key appear in the same subarray. Use $O(1)$ additional space and $O(|A|)$ time.

Problem 6.2, *pg. 53*: *Design a deterministic scheme by which reads and writes to an uninitialized array can be made in $O(1)$ time. You may use $O(n)$ additional storage; reads to uninitialized entry should return* `false`.

Solution 6.2: Create an (uninitialized) array P of n pointers. The array P will maintain a pointer for each initialized entry of A to a back pointer on another array S, itself an (uninitialized) array of n integers. An integer-valued variable t indicates the first empty entry in S; initially, $t = 0$.

Each time entry i from A is to be read, we can check if that entry has been written to before by examining $P[i]$. If $P[i]$ is outside $[0, t-1]$, $A[i]$ is uninitialized. However, even if $P[i]$ is uninitialized, it may lie in $[0, t-1]$. We look at the "back pointer" stored in $S[P[i]]$ and confirm that it is indeed i.

The first time entry i is written in A, we set $S[t]$ to i, $P[i]$ to t and increment t. (We can check that the write is the first write to $A[i]$ by first performing a read and checking if the entry is uninitialized.)

The approach is illustrated in Figure 21.3 on the facing page. The first three entries written are at indices 7, 2, and 1, in that order. Checking if the entry at index 6 is initialized entails examining $P[6]$. If the value in $P[6]$ is not in $[0, 2]$, $A[6]$ is uninitialized. If it is a valid index, e.g., 1, we check if $S[1]$ is 6. For this example, $P[6] = 1$ and $S[P[6]] = 2 \neq 6$, so $A[6]$ is uninitialized.

```
template <typename ValueType, size_t N>
class Array {
  private:
    ValueType A[N];
    int P[N], S[N], t;

    const bool isValid(const size_t &i) const {
      return (0 <= P[i] && P[i] < t && S[P[i]] == i);
```

$t = 3$

	0	1	2	3	4	5	6	7	8	9	10	11
S	7	2	1	?	?	?	?	?	?	?	?	?
P	?	2	1	?	?	?	?	0	?	?	?	?
A	?	✓	✓	?	?	?	?	✓	?	?	?	?

Figure 21.3: Initializing an array in $O(1)$ time.

```
    }

  public:
    Array(void) : t(0) {};

    const bool read(const size_t &i, ValueType &v) const {
      if (isValid(i)) {
        v = A[i];
        return true;
      }
      return false;
    }

    void write(const size_t &i, const ValueType &v) {
      if (!isValid(i)) {
        S[t] = i;
        P[i] = t++;
      }
      A[i] = v;
    }
};
```

Problem 6.3, *pg. 53*: *Design an algorithm that takes a sequence of n three-dimensional coordinates to be traversed, and returns the minimum battery capacity needed to complete the journey. The robot begins with a fully charged battery.*

Solution 6.3: Suppose the three-dimensions correspond to x, y, and z, with z being the vertical dimension. Since energy usage depends on the change in height of the robot, we can ignore the x and y coordinates. Suppose the points where the robot goes in successive order have z coordinates $z_0, \ldots, z_{n-1}$. Assume that the battery capacity is such that with the fully charged battery, the robot can climb B meters. The robot will run out of energy iff there exist integers i and j such that $i < j$ and $z_j - z_i > B$, i.e., to go from Point i to Point j, the robot has to climb more than B meters. Therefore, we would like to pick B such that for any $i < j$, we have $B \geq z_j - z_i$.

We developed several algorithms for this problem in the introduction. Specifically, on Page 2 we showed how to compute the minimum B in $O(n)$ time by keeping the running min as we do a sweep. In code:

```
template <typename HeightType>
```

```
HeightType find_battery_capacity(const vector<HeightType>& h) {
  HeightType min_height = numeric_limits<HeightType>::max(), capacity = 0;
  for (const HeightType &height : h) {
    capacity = max(capacity, height - min_height);
    min_height = min(min_height, height);
  }
  return capacity;
}
```

Problem 6.4, *pg. 54*: *For each of the following, A is an integer array of length n.*

(1.) Compute the maximum value of $(A[j_0] - A[i_0]) + (A[j_1] - A[i_1])$, subject to $i_0 < j_0 < i_1 < j_1$.

(2.) Compute the maximum value of $\sum_{t=0}^{k-1}(A[j_t] - A[i_t])$, subject to $i_0 < j_0 < i_1 < j_1 < \cdots < i_{k-1} < j_{k-1}$. Here k is a fixed input parameter.

(3.) Repeat Problem (2.) when k can be chosen to be any value from 0 to $\lfloor n/2 \rfloor$.

Solution 6.4: The brute-force algorithm for (1.) has complexity $O(n^4)$. The complexity can be improved to $O(n^2)$ by applying the $O(n)$ algorithm to $A[0:j]$ and $A[j+1:n-1]$ for each $j \in [1, n-2]$. However, we can actually solve (1.) in $O(n)$ time by performing a forward iteration and storing the best solution for $A[0:j]$, $j \in [1, n-1]$. We then do a reverse iteration, computing the best solution for $A[j:n-1]$, $j \in [0, n-2]$, which we combine with the result from the forward iteration. The additional space complexity is $O(n)$, which is the space used to store the best solutions for the subarrays.

Here is a straightforward algorithm for (2.). Iterate over j from 1 to k and iterate through A, recording for each index i the best solution for $A[0:i]$ with j pairs. We store these solutions in an auxiliary array of length n. The overall time complexity will be $O(kn^2)$; by reusing the arrays, we can reduce the additional space complexity to $O(n)$.

We can improve the time complexity to $O(kn)$, and the additional space complexity to $O(k)$ as follows. Define B_i^j to be the most money you can have if you must make $j-1$ buy-sell transactions prior to i and buy at i. Define S_i^j to be the maximum profit achievable with j buys and sells with the j-th sell taking place at i. Then the following mutual recurrence holds:

$$\begin{aligned} S_i^j &= A[i] + \max_{i'<i} B_{i'}^j \\ B_i^j &= \max_{i'<i} S_{i'}^{j-1} - A[i] \end{aligned}$$

The key to achieving an $O(kn)$ time bound is the observation that computing B and S requires computing $\max_{i'<i} B_{i'}^{j-1}$ and $\max_{i'<i} S_{i'}^{j-1}$. These two quantities can be computed in constant time for each i and j with a conditional update. In code:

```
template <typename T>
T max_k_pairs_profits(const vector<T>& A, const int &k) {
  vector<T> k_sum(k << 1, numeric_limits<T>::min());
  for (int i = 0; i < A.size(); ++i) {
    vector<T> pre_k_sum(k_sum);
    for (int j = 0, sign = -1; j < k_sum.size() && j <= i; ++j, sign *= -1) {
```

```
      T diff = sign * A[i] + (j == 0 ? 0 : pre_k_sum[j - 1]);
      k_sum[j] = max(diff, pre_k_sum[j]);
    }
  }
  return k_sum.back();  // return the last selling profits as the answer
}
```

Note that the improved solution to (2.) on the preceding page specialized to $k = 2$ strictly subsumes the solution to (1.) on the facing page.

Surprisingly, (3.) on the preceding page can be solved trivially—since we can use an unlimited number of pairs, we can select all pairs $(i, i+1)$ such that $A[i+1] > A[i]$.

Problem 6.5, *pg. 54*: *Design an efficient algorithm for the* 0 mod *n-sum subset problem.*

Solution 6.5: Consider `prefix_sum[j]` $= \sum_{i=0}^{j} A[i] \bmod n$. Either each `prefix_sum[j]` is distinct, in which case for some c we have `prefix_sum[c]` $= 0$ (since `prefix_sum` takes values in $\{0, 1, \ldots, n-1\}$), or for some $a < b$ we have `prefix_sum[a]` $=$ `prefix_sum[b]`.

In the first case, the subarray $A[0:c]$ serves as the result. In the second case, the sum $\sum_{k=a+1}^{b} A[k] \bmod n = 0$, so the subarray $A[a+1:b]$ can be returned as the result.

```
vector<int> find_0_sum_subset(const vector<int> &A) {
  vector<int> prefix_sum(A);
  for (int i = 0; i < prefix_sum.size(); ++i) {
    prefix_sum[i] += i > 0 ? prefix_sum[i - 1] : 0;
    prefix_sum[i] %= A.size();
  }

  vector<int> table(A.size(), -1);
  for (int i = 0; i < A.size(); ++i) {
    if (prefix_sum[i] == 0) {
      vector<int> ans(i + 1);
      iota(ans.begin(), ans.end(), 0);
      return ans;
    } else if (table[prefix_sum[i]] != -1) {
      vector<int> ans(i - table[prefix_sum[i]]);
      iota(ans.begin(), ans.end(), table[prefix_sum[i]] + 1);
      return ans;
    }
    table[prefix_sum[i]] = i;
  }
}
```

Problem 6.6, *pg. 55*: *Design and implement an algorithm that takes as input an array A of n elements, and returns the beginning and ending indices of a longest increasing subarray of A.*

Solution 6.6: The brute-force algorithm is to compute for each i the length m_i of the longest increasing subarray ending at i. This is $m_{i-1} + 1$ if $i \neq 0$ and $A[i-1] < A[i]$, and 1 otherwise. The brute-force algorithm has time complexity $O(n)$, and the space complexity can be reduced to $O(1)$.

We can heuristically improve upon the brute-force algorithm by observing that if $A[i-1] \not< A[i]$ (i.e., we are starting to look for a new subarray starting at i) and the longest contiguous subarray seen up to index i has length L, we can move on to index $i+L$ and work backwards towards i; specifically, if for any $j, i \leq j < i+L$ we have $A[j] \not< A[j+1]$, we can skip the remaining indices.

This is a heuristic in that it does not improve the worst-case complexity—if the array consists of alternating 0s and 1s, we still examine each element—but the best case complexity reduces to $O(\max(n/L, L))$, where L is the length of the longest increasing subarray.

The average case time complexity depends on the probability distribution function for the input, and in general is very difficult to compute. For example, if A is a random permutation, or its entries are independent and uniform in $[0,1]$, it is known that the expected value for L is $\times(\log n/\log\log n)$. If A's entries are independent identically distributed Bernoulli random variables, the longest contiguous nondecreasing subarray has length $\times(\log n)$ in expectation. Both these facts are difficult to prove, and their implications to the average case time complexity are even harder to analyze.

```
template <typename T>
pair<int, int> find_longest_increasing_subarray(const vector<T> &A) {
  int max_len = 1;
  pair<int, int> ans(0, 0);
  int i = 0;
  while (i < A.size()) {
    // Check backwardly and skip if A[j] >= A[j + 1]
    bool is_skippable = false;
    for (int j = i + max_len - 1; j >= i; --j) {
      if (A[j] >= A[j + 1]) {
        i = j + 1;
        is_skippable = true;
        break;
      }
    }

    // Check forwardly if it is not skippable
    if (is_skippable == false) {
      i += max_len - 1;
      while (i + 1 < A.size() && A[i] < A[i + 1]) {
        ++i, ++max_len;
      }
      ans = {i - max_len + 1, i};
    }
  }
  return ans;
}
```

Problem 6.7, *pg. 55*: *How would you compute the weakest implied equivalence relation given n, A, and B? You do not have access to any data structure libraries.*

Solution 6.7: The basic idea is to start by mapping each element to itself. This

mapping is stored in an array F, and can be viewed as implementing a tree relation, with $F[i]$ being i's parent. We iterate through A and B. Since $A[i]$ and $B[i]$ are equivalent, we scan $A[i]$'s ancestors and $B[i]$'s ancestors and update $A[i]$'s (or $B[i]$'s) ancestor to the ancestor which has the smaller index. After all entries in A and B are processed, we make a last pass through F, compressing the ancestor tree, since some parent relationship may have been updated as we iterated through A and B. We return the result as array F; $F[i]$ is the element with the smallest index in the equivalence class of element i.

```
int backtrace(const vector<int> &F, int idx) {
  while (F[idx] != idx) {
    idx = F[idx];
  }
  return idx;
}

/*
 * A and B encode pairwise equivalences on a cardinality N set whose elements
 * are indexed by 0, 1, 2, ..., N-1.
 *
 * For example A[i] = 6 and B[i] = 0 indicates that the 6 and 0 are to be
 * grouped into the same equivalence class.
 *
 * We return the weakest equivalence relation implied by A and B in an array
 * F of length N; F[i] holds the smallest index of all the elements that
 * i is equivalent to.
 */
vector<int> compute_equival_classes(const int &n, const vector<int> &A,
                                    const vector<int> &B) {
  // Each element maps to itself
  vector<int> F(n);
  iota(F.begin(), F.end(), 0);

  for (int i = 0; i < A.size(); ++i) {
    int a = backtrace(F, A[i]), b = backtrace(F, B[i]);
    a < b ? F[b] = a : F[a] = b;
  }

  // Generate the weakest equivalence relation
  for (int &f : F) {
    while (f != F[f]) {
      f = F[f];
    }
  }
  return F;
}
```

Problem 6.8, *pg. 55*: *Suppose you know the permutation σ and the extract sequence $\langle i_0, i_1, \ldots, i_{m-1} \rangle$ in advance. How would you efficiently compute the order in which the m elements are removed from S?*

Solution 6.8: Our algorithm maintains a collection of subsets $\{R_0, R_1, \ldots, R_m\}$ that

partitions $\mathcal{Z}_n$. Specifically, R_k, for $1 \leq k \leq m-1$ consists of elements in σ whose indices are greater than i_{k-1} and less than or equal to i_k. Subset R_0 is all elements in σ with indices less than or equal to i_0. Subset R_m is all elements in σ with indices greater than i_{m-1}. It follows from the definition that $i_{k-1} = i_k$ implies R_k is empty.

We process each $t \in [0, n-1]$ in ascending order. For each t, we determine if it is extracted, and, if it is extracted, when it is extracted. We do this by seeing which R_k it belongs to. If $k = m$, i is never extracted. Otherwise i is removed in the k-th extraction. Consequently, we remove R_k from the partition and add all its elements to the first subset that exists in the partition such that $R_{k'}, k' > k$.

The time complexity is dominated by forming the union of disjoint-sets, and finding the set each of element belongs to. The disjoint-set data structure is ideally suited for union-find and has a run time that is essentially linear.

```
int find_set(vector<int> &set, const int &x) {
  if (set[x] != x) {
    set[x] = find_set(set, set[x]);  // path compression
  }
  return set[x];
}

void union_set(vector<int> &set, const int &x, const int &y) {
  int x_root = find_set(set, x), y_root = find_set(set, y);
  set[min(x_root, y_root)] = max(x_root, y_root);
}

vector<int> offline_minimum(const vector<int> &A, const vector<int> &E) {
  vector<int> R(A.size(), E.size());
  int pre = 0;

  // Initialize the collection of subsets
  for (int i = 0; i < E.size(); ++i) {
    for (int j = pre; j <= E[i]; ++j) {
      R[A[j]] = i;
    }
    pre = E[i] + 1;
  }

  vector<int> ret(E.size(), -1);  // stores the answer
  vector<int> set(E.size() + 1);  // the disjoint-set
  iota(set.begin(), set.end(), 0);  // initializes the disjoint-set
  for (int i = 0; i < A.size(); ++i) {
    if (find_set(set, R[i]) != E.size() && ret[find_set(set, R[i])] == -1) {
      ret[set[R[i]]] = i;
      union_set(set, set[R[i]], set[R[i]] + 1);
    }
  }
  return ret;
}
```

Problem 6.9, *pg. 55*: *Write a function that takes two strings representing integers, and returns an integer representing their product.*

Solution 6.9: We mimic the grade school algorithm for multiplication, i.e., shift, multiply by a digit, and add. The number of digits required for the product is either $n + m$ or $n + m - 1$ for n and m digit operands, so we allocate a string of size $n + m$ for the result; the computation determines whether the number of digits is $n + m$ or $n + m - 1$. We do not store all the partial products, and then add them; rather we add each partial product into the result.

```
class BigInt {
  private:
    int sign;  // -1 or 1;
    vector<char> digits;

  public:
    BigInt(const int &capacity) : sign(1), digits(capacity) {}

    BigInt(const string &s) : sign(s[0] == '-' ? -1 : 1),
                              digits(s.size() - (s[0] == '-')) {
        for (int i = s.size() - 1, j = 0; i >= (s[0] == '-'); --i, ++j) {
          if (isdigit(s[i])) {
            digits[j] = s[i] - '0';
          }
        }
    }

    BigInt operator*(const BigInt &n) const {
      BigInt result(digits.size() + n.digits.size());
      result.sign = sign * n.sign;
      int i, j;
      for (i = 0; i < n.digits.size(); ++i) {
        if (n.digits[i]) {
          int carry = 0;
          for (j = 0; j < digits.size() || carry; ++j) {
            int n_digit = result.digits[i + j] +
                          (j < digits.size() ? n.digits[i] * digits[j] : 0) +
                          carry;
            result.digits[i + j] = n_digit % 10;
            carry = n_digit / 10;
          }
        }
      }

      // If one number is 0, the result size should be 0
      if ((digits.size() == 1 && digits.front() == 0) ||
          (n.digits.size() == 1 && n.digits.front() == 0)) {
        result.digits.resize(1);
      } else {
        result.digits.resize(i + j - 1);
      }
      return result;
    }
};
```

ϵ-Variant 6.9.1: Solve the same problem when numbers are represented as lists of

digits.

Problem 6.10, *pg. 56*: *Given an array A of n elements and a permutation Π, compute $\Pi(A)$ using only constant additional storage.*

Solution 6.10: We can use the fact that every permutation can be expressed as a composition of disjoint cycles, with the decomposition being unique up to ordering.

For example, the permutation $(2, 0, 1, 3)$ can be represented as $(0, 2, 1)(3)$, i.e., we can achieve the permutation $(2, 0, 1, 3)$ by these two moves: $0 \mapsto 2, 2 \mapsto 1, 1 \mapsto 0$, and $3 \mapsto 3$.

If the permutation is presented as a set of disjoint cycles, it can easily be applied using a constant amount of additional storage since we just need to perform rotation by one element. Therefore we want to identify the disjoint cycles that constitute the permutation.

It is straightforward to identify the set of cycles with an additional n bits. Start from any position and keep going forward (from i to $A[i]$) till the initial index is reached, at which point one of the cycles has been found. Then go to another position that is not yet part of any cycle. Finding a position that is not already a part of a cycle is trivial using one bit per array element.

One way to perform this without explicitly using additional $O(n)$ storage is to use the sign bit in the integers that constitute the permutation: Specifically, we subtract n from each entry in `perm` after it has been applied. We check if the element at index i has already been moved by seeing if `perm`$[i]$ is negative.

```
template <typename T>
void apply_permutation1(vector<int> &perm, vector<T> &A) {
  for (int i = 0; i < A.size(); ++i) {
    if (perm[i] >= 0) {
      int a = i;
      T temp = A[i];
      do {
        int next_a = perm[a];
        T next_temp = A[next_a];
        A[next_a] = temp;
        // Mark a as visited by using the sign bit
        perm[a] -= perm.size();
        a = next_a, temp = next_temp;
      } while (a != i);
    }
  }

  // Restore perm back
  size_t size = perm.size();
  for_each(perm.begin(), perm.end(), [size](T &x) { x += size; });
}
```

The code above will apply the permutation in $O(n)$ time but implicitly uses $\Theta(n)$ additional storage, even if it is borrowed from the sign bit of the entries of the `perm` array. We restore `perm` by adding n to each entry after the permutation has been applied.

We can avoid using $\Theta(n)$ additional storage by going from left-to-right and applying the cycle only if the current position is the leftmost position in the cycle. Testing whether the current position is the leftmost position, entails traversing the cycle once more, which increases the run time to $O(n^2)$.

```
template <typename T>
void apply_permutation2(vector<int> &perm, vector<T> &A) {
  for (int i = 0; i < A.size(); ++i) {
    // Traverse the cycle to see if i is the min element
    bool is_min = true;
    int j = perm[i];
    while (j != i) {
      if (j < i) {
        is_min = false;
        break;
      }
      j = perm[j];
    }

    if (is_min) {
      int a = i;
      T temp = A[i];
      do {
        int next_a = perm[a];
        T next_temp = A[next_a];
        A[next_a] = temp;
        a = next_a, temp = next_temp;
      } while (a != i);
    }
  }
}
```

Problem 6.11, *pg. 56*: *Given an array A of integers representing a permutation Π, update A to represent Π^{-1} using only constant additional storage.*

Solution 6.11: The solution is similar to Solution 6.10 on the facing page. All that is needed is to decompose the permutation into a set of cycles and invert each cycle one step back. For example, the permutation $(2, 0, 1, 3)$ can be represented as $(0, 2, 1)(3)$. Hence the inverse can be represented as $(1, 2, 0)(3)$ which amounts to $(1, 2, 0, 3)$.

To save additional space, we can use exactly the same set of tricks as in Solution 6.10 on the preceding page.

Problem 6.12, *pg. 56*: *Given a permutation p represented as a vector, return the vector corresponding to the next permutation under lexicographic ordering. If p is the last permutation, return empty vector. For example, if $p = (1, 0, 3, 2)$, your function should return $(1, 2, 0, 3)$.*

Solution 6.12: The key insight is that if $p[k] < p[k+1]$, and for all $i > k$, $p[i] \geq p[i+1]$, then no permutation of the elements consequent to k will lead to a permutation that is ahead of p in the lexicographic order. Therefore, we must increase $p[k]$. To

obtain the next permutation we find the largest index l such that $p[l] > p[k]$ (such an l must exist since $p[k] < p[k+1]$). Swapping $p[l]$ and $p[k]$ leaves the sequence after position k in decreasing order. Reversing this sequence after position k produces its lexicographically minimal permutation, and the lexicographic successor of the original p.

To find the previous permutation, we apply the same idea with some modifications.

```
vector<int> next_permutation(vector<int> p) {
  int k = p.size() - 2;
  while (k >= 0 && p[k] >= p[k + 1]) {
    --k;
  }
  if (k == -1) {
    return {};  // p is the last permutation
  }

  int l;
  for (int i = k + 1; i < p.size(); ++i) {
    if (p[i] > p[k]) {
      l = i;
    } else {
      break;
    }
  }
  swap(p[k], p[l]);

  // Produce the lexicographically minimal permutation
  reverse(p.begin() + k + 1, p.end());
  return p;
}
```

Variant 6.12.1: Compute the k-th permutation under lexicographic ordering, starting from the identity permutation, which is the first permutation in lexicographic ordering.

ϵ-Variant 6.12.2: Given a permutation p represented as a vector, return the vector corresponding to the *previous* permutation of p under lexicographic ordering.

Problem 6.13, *pg. 56*: *Design a $\Theta(n)$ algorithm for rotating an array A of n elements to the right by i positions. You are allowed $O(1)$ additional storage.*

Solution 6.13: This is a special case of applying a permutation with constant additional storage (Problem 6.10 on Page 56) with the permutation corresponding to a rotation. A rotation corresponds to a set of cycles of the form $\langle c, (i+c) \bmod n, (2i+c) \bmod n, \ldots, (mi+c) \bmod n \rangle$ for a number of different values of c. For example, for the case where $n = 6$ and $i = 2$, the corresponding cycles are $(0, 2, 4)$ and $(1, 3, 5)$. When $n = 15$ and $i = 6$, the cycles are $\langle 0, 6, 12, 3, 9 \rangle$, $\langle 1, 7, 13, 4, 10 \rangle$, and $\langle 2, 8, 14, 5, 11 \rangle$.

These examples lead us to conjecture the following:

(1.) All cycles have the same length, and are a shifted version of the cycle $\langle 0, i \bmod n, 2i \bmod n, \ldots (l-1)i \bmod n \rangle$.

(2.) The number of cycles is the GCD of n and i.

These conjectures can be justified on heuristic grounds, specifically from considering the prime factorizations for i and n. See on the next page for the formal proof.

Assuming these conjectures to be correct, we can apply the rotation one cycle at a time, as follows. The first elements of the different cycles are at indices $0, 1, 2, \ldots, \text{GCD}(n, i) - 1$. For each cycle, we assign the index of first element to a temporary variable j. We iteratively move the element at j to $(j+i) \bmod n$ and update j to $(j+i) \bmod n$, stopping after $n/\text{GCD}(n, i)$ moves. This takes $O(1)$ space: a variable to track which cycle we are processing, a variable to track how many elements we have processed in the current cycle, as well as temporary variables for performing the move.

```
template <typename T>
void rotate_array(vector<T> &A, int i) {
  i %= A.size();
  int cycles = GCD(A.size(), i);  // number of cycles in this rotation
  int hops = A.size() / cycles;  // number of elements in a cycle

  for (int c = 0; c < cycles; ++c) {
    T temp = A[c];
    for (int j = 1; j < hops; ++j) {
      swap(A[(c + j * i) % A.size()], temp);
    }
    A[c] = temp;
  }
}
```

We now provide an alternative to the permutation approach. The new solution works well in practice and is considerably simpler. Assume that $A = \langle 1, 2, 3, 4, a, b \rangle$, and $i = 2$. Then in the rotated A there are two subarrays, $\langle 1, 2, 3, 4 \rangle$ and $\langle a, b \rangle$ that keep their original orders. Therefore, rotation can be seen as the exchanges of the two subarrays of A. To achieve these exchanges using only $O(1)$ space we use a `reverse` function. Using A and i as an example, we first reverse A to get A' ($\langle 1, 2, 3, 4, a, b \rangle \mapsto \langle b, a, 4, 3, 2, 1 \rangle$), then reverse the first i elements of A' ($\langle b, a, 4, 3, 2, 1 \rangle \mapsto \langle a, b, 4, 3, 2, 1 \rangle$), and reverse the remaining elements starting from the i-th element of A' ($\langle a, b, 4, 3, 2, 1 \rangle \mapsto \langle a, b, 1, 2, 3, 4 \rangle$) which yields the rotated A. Following is the code in C++:

```
template <typename T>
void rotate_array(vector<T> &A, int i) {
  i %= A.size();
  reverse(A.begin(), A.end());
  reverse(A.begin(), A.begin() + i);
  reverse(A.begin() + i, A.end());
}
```

We now prove Conjectures (1.) and (2.).

Proof:

First we prove that rotation does result in cycles. Take l_0 to be the largest integer such that the sequence $\sigma_0 = \langle 0, i \bmod n, 2i \bmod n, 3i \bmod n, \ldots, ((l_0 - 1)i) \bmod n \rangle$ does not repeat. We claim that $(l_0 i) \bmod n = 0$. Since l_0 was defined to be maximal, it must be that $(l_0 i) \bmod n$ is a value that is already in σ_0. For contradiction, suppose it equals $\sigma_0(r), 0 < r < (l_0 - 1)$. Then

$$\begin{aligned}
\sigma_0(l_0 - r) \bmod n &= ((l_0 - r)i) \bmod n \\
&= ((l_0 i) \bmod n - (ri) \bmod n) \bmod n \\
&= ((l_0 i) \bmod n - \sigma_0(r) \bmod n) \bmod n \\
&= 0 \bmod n.
\end{aligned}$$

Hence the sequence repeats at $\sigma_0(l_0 - r)$, contradicting the maximality of l_0.

Define l_c to be the largest integer such that the sequence $\sigma_c = \langle c, (i + c) \bmod n, (2i + c) \bmod n, (3i + c) \bmod n, \ldots, ((l_c - 1)i + c) \bmod n \rangle$ does not repeat. Conjecture (1.) on the preceding page, namely that all cycles have the same length, follows from the observation that the difference between $(ij) \bmod n$ and $(ij + c) \bmod n$ always equals $c \bmod n$.

Now we prove Conjecture (2.) on the previous page, i.e., there exist exactly $\mathrm{GCD}(n, i)$ cycles. Since we have just seen that all cycles have the same length, it suffices to prove that the length of the cycle containing 0 is $n/\mathrm{GCD}(n, i)$.

Let g be the smallest integer greater than 0 that appears in the cycle which contains 0. Because of the modulus operation, g is not necessarily the number that follows 0, e.g., when $n = 15$ and $i = 6$, g is 3, even though the cycle corresponding to 0 is $\langle 0, 6, 12, 3, 9 \rangle$. The set S_0 of numbers in the cycle that 0 belongs to is $\{x \mid \exists j\ x = ij \bmod n\}$. Equivalently, $S_0 = \{x \mid \exists a \exists b\ x = (ai + bn) \bmod n\}$. It is a basic fact that the GCD of i and n is the smallest positive integer of the form $ai + bn$, with a and b being arbitrary integers. Therefore g is the GCD of n and i.

We claim that S_0 is exactly equal to the set of numbers in $\{0, 1, \ldots, n - 1\}$ that are divisible by g. Conjecture (2.) follows from the fact that exactly $n/\mathrm{GCD}(n, i)$ numbers in $\{0, 1, \ldots, n - 1\}$ are divisible by g.

First we prove that all numbers in S_0 are divisible by g. If not, say $e = ij \bmod n$ is not divisible by g. Then $e = gq + r$, where $r \in (0, g)$ is the remainder. Since $g = ai + bn$ for some a and b, we have $r = (e - (ai + bn)) \bmod n = (e - ai) \bmod n$. Since e lies in S_0, all numbers of the form $(e + Gi) \bmod n$, where $G \geq 0$, also lie in S_0. In particular, let H be such that $Hn - a \geq 0$. Then $(e + (Hn - a)i) \bmod n$ lies in S_0. But $(e + (Hn - a)i) \bmod n = (e - ai) \bmod n = r$, which contradicts the minimality of g.

Now we show that $gI \bmod n \in S_0$ for all I. Since $g = ai + bn$, for some a and b, we have $gI \bmod n = (aiI + bnI) \bmod n = aIi \bmod n$. Let J be such that $(Jn + aI) \geq 0$. Then $(Jn + aI)i \bmod n$ is in S_0. But $(Jn + aI)i \bmod n = aIi \bmod n = gI \bmod n$, demonstrating that $gI \bmod n \in S_0$.

Problem 6.14, *pg. 57*: *Check whether a 9×9 2D array representing a partially completed Sudoku is valid. Specifically, check that no row, column, and 3×3 2D subarray contains duplicates. A 0-value in the 2D array indicates that entry is blank; every other entry is in* $[1, 9]$.

Solution 6.14: We need to check nine row constraints, nine column constraints, and nine sub-grid constraints. We use bit arrays to test for constraint violations, that is to ensure no number in $[1, 9]$ appears more than once.

```
// Check if a partially filled matrix has any conflicts
bool is_valid_Sudoku(const vector<vector<int>> &A) {
  // Check row constraints
  for (int i = 0; i < A.size(); ++i) {
    vector<bool> is_present(A.size() + 1, false);
    for (int j = 0; j < A.size(); ++j) {
      if (A[i][j] != 0 && is_present[A[i][j]] == true) {
        return false;
      } else {
        is_present[A[i][j]] = true;
      }
    }
  }

  // Check column constraints
  for (int j = 0; j < A.size(); ++j) {
    vector<bool> is_present(A.size() + 1, false);
    for (int i = 0; i < A.size(); ++i) {
      if (A[i][j] != 0 && is_present[A[i][j]] == true) {
        return false;
      } else {
        is_present[A[i][j]] = true;
      }
    }
  }

  // Check region constraints
  int region_size = sqrt(A.size());
  for (int I = 0; I < region_size; ++I) {
    for (int J = 0; J < region_size; ++J) {
      vector<bool> is_present(A.size() + 1, false);
      for (int i = 0; i < region_size; ++i) {
        for (int j = 0; j < region_size; ++j) {
          if (A[region_size * I + i][region_size * J + j] != 0 &&
              is_present[A[region_size * I + i][region_size * J + j]]) {
            return false;
          } else {
            is_present[A[region_size * I + i][region_size * J + j]] = true;
          }
        }
      }
    }
  }
  return true;
}
```

Solution 17.8 on Page 395 describes how to solve Sudoku instances using branch and bound.

Problem 6.15, *pg. 57*: *Implement a function which takes a 2D array A and prints A in spiral order.*

Solution 6.15: The outermost elements of an $n \times n$ 2D array can be written in spiral order using four iterations: elements $(0,0)$ to $(0,n-2)$, then elements $(0,n-1)$ to $(n-2,n-1)$, followed by elements $(n-1,n-1)$ to $(n-1,1)$, and finally elements $(n-1,0)$ to $(1,0)$. After this, we are left with the problem of printing the elements of an $(n-2) \times (n-2)$ 2D array in spiral order. This leads to an iterative algorithm that prints the outermost elements of $n \times n, (n-2) \times (n-2), (n-4) \times (n-4), \ldots$ 2D arrays.

```
void print_matrix_clockwise(const vector<vector<int>> &A, const int &offset) {
  if (offset == A.size() - offset - 1) {  // for matrix with odd size
    cout << A[offset][offset];
  }

  for (int j = offset; j < A.size() - offset - 1; ++j) {
    cout << A[offset][j] << ' ';
  }
  for (int i = offset; i < A.size() - offset - 1; ++i) {
    cout << A[i][A.size() - offset - 1] << ' ';
  }
  for (int j = A.size() - offset - 1; j > offset; --j) {
    cout << A[A.size() - offset - 1][j] << ' ';
  }
  for (int i = A.size() - offset - 1; i > offset; --i) {
    cout << A[i][offset] << ' ';
  }
}

void print_matrix_in_spiral_order(const vector<vector<int>> &A) {
  for (int offset = 0; offset < ceil(0.5 * A.size()); ++offset) {
    print_matrix_clockwise(A, offset);
  }
}
```

An alternate solution in C++ writes 0 into array entries to indicate they have been processed, and a `shift` 2D array to compress the four iterations above into a single iterations parametrized by `shift`:

```
void print_matrix_spiral(vector<vector<int>> A) {
  const array<array<int, 2>, 4> shift = {0, 1, 1, 0, 0, -1, -1, 0};
  int dir = 0, x = 0, y = 0;

  for (int i = 0; i < A.size() * A.size(); ++i) {
    cout << A[x][y] << ' ';
    A[x][y] = 0;
    int nx = x + shift[dir][0], ny = y + shift[dir][1];
    if (nx < 0 || nx >= A.size() || ny < 0 || ny >= A.size() ||
        A[nx][ny] == 0) {
      dir = (dir + 1) & 3;
```

```
      nx = x + shift[dir][0], ny = y + shift[dir][1];
    }
    x = nx, y = ny;
  }
}
```

ϵ-Variant 6.15.1: Given a dimension d, write a program to generate a $d \times d$ 2D array which when printed in spiral order outputs the sequence $\langle 1, 2, 3, \ldots, d^2 \rangle$. For example, if $d = 3$, the result should be

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 8 & 9 & 4 \\ 7 & 6 & 5 \end{bmatrix}.$$

ϵ-Variant 6.15.2: Given a sequence of integers σ, compute a 2D array A which when printed in spiral order yields σ. (Assume $|\sigma| = n^2$ for some integer n.)

ϵ-Variant 6.15.3: Write a program to enumerate the first n pairs of integers (a, b) in spiral order, starting from $(0, 0)$ followed by $(1, 0)$. For example, if $n = 10$, your output should be $(0, 0), (1, 0), (1, -1), (0, -1), (-1, -1), (-1, 0), (-1, 1), (0, 1), (1, 1), (2, 1)$.

Problem 6.16, *pg. 58*: *Implement a routine that takes a $D \times D$ Boolean array A together with an entry (x, y) and flips the color of the region associated with (x, y). See Figure 6.5 on Page 58 for an example of flipping.*

Solution 6.16: Conceptually, we solve this problem by maintaining a queue `q` of entries to process, and a 2D Boolean array `processed` indicating whether an entry has been processed. Initially, all entries in `processed` are marked `false` and `q` contains (x, y). The queue is popped iteratively, and the neighbors of the popped element are examined. Any neighbor which is unprocessed and whose color needs to be changed is added to `q`. After its neighbors have been examined, the processed status of the element just popped is set to `true` and its color is flipped. The computation ends when `q` is empty.

In practice, we do not require the `processed` array, since there are only two colors. We only need to record the color of the initial entry, and compare new entries with that color.

```
void flip_color(vector<vector<bool>> &A, const int &x, const int &y) {
  const array<array<int, 2>, 4> dir = {-1, 0, 1, 0, 0, -1, 0, 1};
  const bool color = A[x][y];

  queue<pair<int, int>> q;
  q.emplace(x, y);
  while (q.empty() == false) {
    pair<int, int> curr(q.front());
    A[curr.first][curr.second] = !A[curr.first][curr.second];  // flip color
    for (auto &d : dir) {
```

```
      pair<int, int> next(curr.first + d[0], curr.second + d[1]);
      if (next.first >= 0 && next.first < A.size() &&
          next.second >= 0 && next.second < A[next.first].size() &&
          A[next.first][next.second] == color) {
        q.emplace(next);
      }
    }
    q.pop();
  }
}
```

We also provide a recursive solution which does not need a queue but implicitly uses a stack. This solution does not require a processed array; instead, we temporarily flip the color of the entry being processed.

```
void flip_color(vector<vector<bool>> &A, const int &x, const int &y) {
  const array<array<int, 2>, 4> dir = {-1, 0, 1, 0, 0, -1, 0, 1};
  const bool color = A[x][y];
  A[x][y] = !A[x][y];  // flip the color

  for (auto &d : dir) {
    const int nx = x + d[0], ny = y + d[1];
    if (nx >= 0 && nx < A.size() && ny >= 0 && ny < A[nx].size() &&
        A[nx][ny] == color) {
      flip_color(A, nx, ny);
    }
  }
}
```

ϵ-Variant 6.16.1: Design an algorithm for computing the black region that contains the most points.

ϵ-Variant 6.16.2: Design an algorithm that takes a point (a, b), sets $A(a, b)$ to black, and returns the size of the black region that contains the most points. Assume this algorithm will be called multiple times, and you want to keep the aggregate run time as low as possible.

Problem 6.17, *pg. 58*: *Design an algorithm that rotates a $n \times n$ 2D array by 90 degrees clockwise. Assume that $n = 2^k$ for some positive integer k. What is the time complexity of your algorithm?*

Solution 6.17: It is natural to use recursion: decompose the 2D array into four equal-sized subarrays, $A[0 : \frac{n}{2} - 1][0 : \frac{n}{2} - 1]$, $A[0 : \frac{n}{2} - 1][\frac{n}{2} : n - 1]$, $A[\frac{n}{2} : n - 1][0 : \frac{n}{2} - 1]$, and $A[\frac{n}{2} : n - 1][\frac{n}{2} : n - 1]$, and recursively rotate each of these. Consequently, make a copy C of $A[0 : \frac{n}{2} - 1][0 : \frac{n}{2} - 1]$, and copy $A[0 : \frac{n}{2} - 1][\frac{n}{2} : n - 1]$ into $A[0 : \frac{n}{2} - 1][0 : \frac{n}{2} - 1]$, $A[\frac{n}{2} : n - 1][\frac{n}{2} : n - 1]$ into $A[0 : \frac{n}{2} - 1][\frac{n}{2} : n - 1]$, $A[\frac{n}{2} : n - 1][0 : \frac{n}{2} - 1]$ into $A[\frac{n}{2} : n - 1][\frac{n}{2} : n - 1]$ and C into $A[\frac{n}{2} : n - 1][0 : \frac{n}{2} - 1]$. The run time satisfies the recurrence $T(n) = 4T(\frac{n}{4}) + O(n)$, which solves to $O(n \log n)$, where $n = 2^{2k}$.

```
template <typename T>
void copy_matrix(vector<vector<T>> &A, const int &A_x_s, const int &A_x_e,
                 const int &A_y_s, const int &A_y_e,
                 const vector<vector<T>> &S,
                 const int &S_x, const int &S_y) {
  for (int i = 0; i < A_x_e - A_x_s; ++i) {
    copy(S[S_x + i].cbegin() + S_y, S[S_x + i].cbegin() + S_y + A_y_e - A_y_s,
         A[A_x_s + i].begin() + A_y_s);
  }
}

template <typename T>
void rotate_matrix_helper(vector<vector<T>> &A, const int &x_s,
                          const int &x_e, const int &y_s, const int &y_e) {
  if (x_e > x_s + 1) {
    int mid_x = x_s + ((x_e - x_s) >> 1), mid_y = y_s + ((y_e - y_s) >> 1);
    // Move submatrices
    vector<vector<T>> C(mid_x - x_s, vector<T>(mid_y - y_s));
    copy_matrix(C, 0, C.size(), 0, C.size(), A, x_s, y_s);
    copy_matrix(A, x_s, mid_x, y_s, mid_y, A, mid_x, y_s);
    copy_matrix(A, mid_x, x_e, y_s, mid_y, A, mid_x, mid_y);
    copy_matrix(A, mid_x, x_e, mid_y, y_e, A, x_s, mid_y);
    copy_matrix(A, x_s, mid_x, mid_y, y_e, C, 0, 0);

    // Recursively rotate submatrices
    rotate_matrix_helper(A, x_s, mid_x, y_s, mid_y);
    rotate_matrix_helper(A, x_s, mid_x, mid_y, y_e);
    rotate_matrix_helper(A, mid_x, x_e, mid_y, y_e);
    rotate_matrix_helper(A, mid_x, x_e, y_s, mid_y);
  }
}

template <typename T>
void rotate_matrix(vector<vector<T>> &A) {
  rotate_matrix_helper(A, 0, A.size(), 0, A.size());
}
```

Alternately, we could perform the rotation using $O(1)$ additional memory in $O(n)$ time by iterating through any one of the four subarrays, and rotating elements in sets of four.

Variant 6.17.1: Suppose the underlying hardware has support for fast two-dimensional block copies. Specifically, you can copy an $m \times m$ 2D array in $O(m)$ time. How can you exploit the hardware to reduce the time complexity?

Problem 6.18, *pg. 59*: *Implement run-length encoding and decoding functions. Assume the string to be encoded consists of letters of the alphabet, with no digits, and the string to be decoded is a valid encoding.*

Solution 6.18: The decoding function entails converting a number represented in decimal to its integer equivalent; the encoding function entails the reverse. Both of these are covered in Solution 5.6 on Page 176. The remainder of the code consists of

iterating through the input string and appending to the result string.

```
string decoding(const string &s) {
  int count = 0;
  string ret;
  for (const char &c : s) {
    if (isdigit(c)) {
      count = count * 10 + c - '0';
    } else {  // isalpha
      ret.append(count, c);
      count = 0;
    }
  }
  return ret;
}

string encoding(const string &s) {
  int count = 1;
  stringstream ss;
  for (int i = 1; i < s.size(); ++i) {
    if (s[i] == s[i - 1]) {
      ++count;
    } else {
      ss << count << s[i - 1];
      count = 1;
    }
  }
  ss << count << s.back();
  return ss.str();
}
```

Problem 6.19, *pg. 59*: *Implement a function for reversing the words in a string. Your function should use $O(1)$ space.*

Solution 6.19: The code for computing the position for each character in a single pass is fairly complex. However, a two stage iteration is easy. In the first step, reverse the entire string and in the second step, reverse each word. For example, "ram is costly" transforms to "yltsoc si mar", which transforms to "costly is ram". Here is code in C++:

```
void reverse_words(string &input) {
  // Reverse the whole string first
  reverse(input.begin(), input.end());

  size_t start = 0, end;
  while ((end = input.find(" ", start)) != string::npos) {
    // Reverse each word in the string
    reverse(input.begin() + start, input.begin() + end);
    start = end + 1;
  }
  // Reverse the last word
  reverse(input.begin() + start, input.end());
}
```

Problem 6.20, *pg. 59*: *Given two strings s (the "search string") and t (the "text"), find the first occurrence of s in t.*

Solution 6.20: Knuth-Morris-Pratt, Boyer-Moore, and Rabin-Karp. are widely taught algorithms for substring search that run in linear time. In practice Boyer-Moore is the fastest string search algorithm, because for many applications, it runs in sub-linear time.

The Boyer-Moore algorithm works by trying to match characters of s in t at a certain offset in the reverse order (last character of s matched first). If we can match all the characters in s, then we have found a match; otherwise, we stop at the first mismatch. The key idea behind the Boyer-Moore algorithm is to skip as many offsets as possible when we are done matching characters at a given offset. We do this by building two tables—the good suffix shift table and the bad character shift table.

For a given character, the bad character shift table gives us the distance of the last occurrence of that character in s to the rightmost string. If the character does not occur in s, then the entry in the table is $|s|$. Hence when we find a character in t that does not match for the current offset, we know how much we must move forward so that this character can match for the first time.

The good suffix shift table is a little more complex. Conceptually, for a given suffix x of s, it tells us what is the shortest suffix y of s that is longer than x and has x as suffix. In practice what we store is how far can we move safely, given that we have matched up to $|x|$ characters but did not match the next character.

The Rabin-Karp algorithm is based on the idea of "fingerprinting". It compute hash codes of each substring $t[i : i+|s|-1]$ for $i = 0$ to $|t|-|s|$—these are the fingerprints. If $h(t[i : i+|s|-1]) \neq h(s)$, the $|s|$ length substring beginning at i cannot equal s. A good hash function is one where the probability of collisions is low and $h(t[i : i+|s|-1])$ can be incrementally computed, that is the time to compute $h(t[i : i+|s|-1])$, given $h(t[i-1 : i+|s|-2])$, is $O(1)$. (Such a hash function is sometimes referred to as a rolling hash.) The Rabin-Karp algorithm is very simple to implement, and generalizes more easily, e.g., to two dimension pattern matching, than the Knuth-Morris-Pratt and Boyer-Moore algorithms. The expected time complexity is $O(|s| + |t|)$.

```
const int base = 26, mod = 997;

int rabin_karp(const string &t, const string &s) {
  if (s.size() > t.size()) {
    return -1;  // s is not a substring of t
  }

  int t_hash = 0, s_hash = 0;
  for (int i = 0; i < s.size(); ++i) {
    t_hash = (t_hash * base + t[i]) % mod;
    s_hash = (s_hash * base + s[i]) % mod;
  }

  for(int i = s.size(); i < t.size(); ++i) {
    // In case of hash collision, check the two substrings are actually equal
    if (t_hash == s_hash && t.compare(i - s.size(), s.size(), s) == 0) {
```

```
      return i - s.size();  // find match
    }
    t_hash -= (t[i - s.size()] * static_cast<int>(pow(base, s.size() - 1)))
              % mod;
    if (t_hash < 0) {
      t_hash += mod;
    }
    t_hash = (t_hash * base + t[i]) % mod;
  }

  if (t_hash == s_hash && t.compare(t.size() - s.size(), s.size(), s) == 0) {
    return t.size() - s.size();
  }
  return -1;  // s is not a substring of t
}
```

Problem 6.21, *pg. 60*: *Write a function which takes as input a string s, and removes each "b" and replaces each "a" by "dd". Use $O(1)$ additional storage—assume s is stored in an array that has enough space for the final result.*

Solution 6.21: We start by making a first pass through s in which we delete each "b" by maintaining a write index, `write_idx` and a current index, `cur_idx`—we achieve the effect of deleting "b" by skipping over "b". We also count the number of "a"s. We then make a second pass working backwards from the end of the current string, copying characters to the end of the resulting string (whose size we know from the number of "a"s). For each "a", we write "dd".

```
string replace_and_remove(string s) {
  // Remove "b" and count the number of "a"
  int write_idx = 0, a_count = 0;
  for (const char &c : s) {
    if (c != 'b') {
      s[write_idx++] = c;
    }
    if (c == 'a') {
      ++a_count;
    }
  }

  // Allocate space according to the number of "a"
  s.resize(write_idx + a_count);
  // Replace "a" with "dd";
  int cur_idx = write_idx - 1;
  write_idx = s.size() - 1;
  while (cur_idx >= 0) {
    if (s[cur_idx] == 'a') {
      s[write_idx--] = s[write_idx--] = 'd';
    } else {
      s[write_idx--] = s[cur_idx];
    }
    --cur_idx;
  }
  return s;
```

```
}
```

We can prove that the second step correctly replaces each "a" by "dd" by induction on the length of the string n.

Proof:

For the base case, i.e., length 1 string, there are two possibilities—the string is "a" or x, where x is one of {"b", "c", "d"}. For both possibilities, induction goes through. Assume by induction that the construction is correct for all strings of length $n > 1$. Consider a string s of length $n + 1$. Let's say the length of the final result is k. If s ends in "c" or "d", we copy $s[n-1]$ over to $s[k-1]$. By the induction hypothesis, our construction correctly copies the substring s^{n-1} consisting of the first $n-1$ characters of s to the remaining $k-1$ locations. If s ends in "a", we write "d" into locations $k-2$ and $k-1$. Now we have to process s^{n-1}, which will require $k-2$ locations. By the induction hypothesis, the construction correctly writes the result into these locations, and induction goes through.

ϵ-Variant 6.21.1: You have an array C of characters. The characters may be letters, digits, blanks, and punctuation. The telex-encoding of the array C is an array T of characters in which letters, digits, and blanks appear as before, but punctuation marks are spelled out. For example, telex-encoding entails replacing the character "." by the string "DOT", the character "," by "COMMA", the character "?" by "QUESTION MARK", and the character "!" by "EXCLAMATION MARK". Design an algorithm to perform telex-encoding with $O(1)$ space.

Variant 6.21.2: Write a function which merges two sorted arrays of integers, A and B. Specifically, the final result should be a sorted array of length $|A| + |B|$. Use $O(1)$ additional storage—assume the result is stored in A, which has sufficient space. These arrays are C-style arrays, i.e., contiguous preallocated blocks of memory.

Problem 6.22, *pg. 60*: *Given a cell phone keypad (specified by a mapping M that takes individual digits and returns the corresponding set of characters) and a number sequence, return all possible character sequences (not just legal words) that correspond to the number sequence.*

Solution 6.22: Recursion is natural. Let P be an n-digit number sequence. Assume these digits are indexed starting at 0, i.e., $P[0]$ is the first digit. Let S be a character sequence corresponding to the first k digits of P. We can generate all length n character sequences corresponding to P that have S as their prefix as follows. If $k = n$, there is nothing to do. Otherwise, we recurse on each length-$k + 1$ sequence of the form Sx, for each $x \in M(P[k])$.

```
const array<string, 10> M = {"0", "1", "ABC", "DEF", "GHI", "JKL", "MNO",
                             "PQRS", "TUV", "WXYZ"};

void phone_mnemonic_helper(const string &num, const int &d, string &ans) {
  if (d == num.size()) {
```

```
    cout << ans << endl;
  } else {
    for (const char &c : M[num[d] - '0']) {
      ans[d] = c;
      phone_mnemonic_helper(num, d + 1, ans);
    }
  }
}

void phone_mnemonic(const string &num) {
  string ans(num.size(), 0);
  phone_mnemonic_helper(num, 0, ans);
}
```

Problem 6.23, *pg. 60*: *Design an algorithm that takes a string s and a string r, assumed to be a well-formed ESRE, and checks if r matches s.*

Solution 6.23: The key to solving this problem is using recursion effectively.

If r starts with ^, then the remainder of r, i.e., r^1, must strictly match a prefix of s. If r ends with a $, some suffix of s must be strictly matched by r without the trailing $. Otherwise, r must strictly match some substring of s.

Call the function that checks whether r strictly matches a prefix of string s `is_match`. This function has to check several cases:

(1.) Length-0 ESREs which match everything.
(2.) An ESRE starting with ^ or ending with $.
(3.) An ESRE starting with an alphanumeric character or dot.
(4.) An ESRE starting with a * match, e.g., a*wXY or .*Wa.

Case (1.) is a base case. Case (2.) involves a check possibly followed by a recursive call to `is_match_here`. Case (3.) requires a single call to `is_match_here`. Case (4.) is handled by a walk down the string s, checking that the prefix of s thus far matches the alphanumeric character or dot until some suffix of s is matched by the remainder of the ESRE, i.e., r^2.

```
bool is_match_here(const string &r, const string &s) {
  // Case (1.)
  if (r.empty()) {
    return true;
  }

  // Case (2) : ends with '$'
  if (r.front() == '$' && r.size() == 1) {
    return s.empty();
  }

  // Case (4.)
  if (r.size() >= 2 && r[1] == '*') {
    for (int i = 0; i < s.size() && (r.front() == '.' || r.front() == s[i]);
         ++i) {
      if (is_match_here(r.substr(2), s.substr(i + 1))) {
        return true;
      }
```

```
    }
    return is_match_here(r.substr(2), s);
  }

  // Case (3.)
  return !s.empty() && (r.front() == '.' || r.front() == s.front()) &&
         is_match_here(r.substr(1), s.substr(1));
}

bool is_match(const string &r, const string &s) {
  // Case (2.) : starts with '^'
  if (r.front() == '^') {
    return is_match_here(r.substr(1), s);
  }

  for (int i = 0; i <= s.size(); ++i) {
    if (is_match_here(r, s.substr(i))) {
      return true;
    }
  }
  return false;
}
```

ϵ-Variant 6.23.1: Solve the same problem for regular expressions without the ^ and $ operators.

Problem 7.1, *pg. 63*: *Write a function that takes L and F, and returns the merge of L and F. Your code should use $O(1)$ additional storage—it should reuse the nodes from the lists provided as input. Your function should use $O(1)$ additional storage, as illustrated in Figure 7.3 on Page 63. The only field you can change in a node is* `next`.

Solution 7.1: We traverse the lists, using one pointer per list, each initialized to the list head. We compare the contents of the pointer—the pointer with the lesser contents is to be added to the end of the result and advanced. If either pointer is `null`, we add the sublist pointed to by the other to the end of the result. The add can be performed by a single pointer update—it does not entail traversing the sublist. The worst case time complexity corresponds to the case when the lists are of comparable length. In the best case, one list is much shorter than the other and all its entries appear at the beginning of the merged list.

```
template <typename T>
void append_node(shared_ptr<node_t<T>> &head, shared_ptr<node_t<T>> &tail,
                 shared_ptr<node_t<T>> &n) {
  head ? tail->next = n : head = n;
  tail = n;  // reset tail to the last node
}

template <typename T>
void append_node_and_advance(shared_ptr<node_t<T>> &head,
                             shared_ptr<node_t<T>> &tail,
                             shared_ptr<node_t<T>> &n) {
```

```
  append_node(head, tail, n);
  n = n->next;  // advance n
}

template <typename T>
shared_ptr<node_t<T>> merge_sorted_linked_lists(shared_ptr<node_t<T>> F,
                                                shared_ptr<node_t<T>> L) {
  shared_ptr<node_t<T>> sorted_head = nullptr, tail = nullptr;

  while (F && L) {
    append_node_and_advance(sorted_head, tail, F->data < L->data ? F : L);
  }

  // Append the remaining nodes of F
  if (F) {
    append_node(sorted_head, tail, F);
  }
  // Append the remaining nodes of L
  if (L) {
    append_node(sorted_head, tail, L);
  }
  return sorted_head;
}
```

ϵ-Variant 7.1.1: Solve the same problem when the lists are doubly linked.

Problem 7.2, *pg. 63*: *Given a reference to the head of a singly linked list L, how would you determine whether L ends in a* `null` *or reaches a cycle of nodes? Write a function that returns* `null` *if there does not exist a cycle, and the reference to the start of the cycle if a cycle is present. (You do not know the length of the list in advance.)*

Solution 7.2: This problem has several solutions. If space is not an issue, the simplest approach is to explore nodes via the next field starting from the head and storing visited nodes in a hash table—a cycle exists iff we visit a node already in the hash table. If no cycle exists, the search ends at the tail (often represented by having the next field set to `null`). This solution requires $\Theta(n)$ space, where n is the number of nodes in the list.

In some languages, e.g., C, the next field is a pointer. Typically, for performance reasons related to the memory subsystem on a processor, memory is allocated on word boundaries, and (at least) two of the least significant bits in the next pointer are 0. Bit fiddling can be used to set the least significant bit on the next pointer to mark whether a node as been visited. This approach has the disadvantage of changing the data structure—these updates can be undone later.

Another approach is to reverse the linked list, in the manner of Solution 7.9 on Page 215. If the head is encountered during the reversal, it means there is a cycle; otherwise we will get to the tail. Although this approach requires no additional storage, and runs in $O(n)$ time, it does modify the list.

A naïve approach that does not use additional storage and does not modify the

list is to walk the list in two loops—the outer loop visits the nodes one-by-one, and the inner loop starts from the head, and visits m nodes, where m is the number of nodes visited in the outer loop. If the node being visited by the outer loop is visited twice, a loop has been detected. (If the outer loop encounters the end of the list, no cycle exists.) This approach has $O(n^2)$ time complexity.

This idea can be made to work in linear time—use a slow pointer, `slow`, and a fast pointer, `fast`, to visit the list. In each iteration, advance `slow` by one and `fast` by two. The list has a cycle iff the two pointers meet.

This is proved as follows.

Proof:

Number the nodes in the cycle by assigning first node encountered the index 0. Let C be the total number of nodes in the cycle. If the fast pointer reaches the first node at iteration F, at iteration $i \geq F$, it will be at node $2(i - F) \bmod C$. If the slow pointer reaches the first node at iteration S, at iteration $i \geq S$, it will be at node $(i - S) \bmod C$. The difference between the pointer locations after the slow pointer reaches the first node in the cycle is $2(i - F) - (i - S) \bmod C = i - (2F - S) \bmod C$. As i increases by one in each iteration, the equation $(i - (2F - S)) \bmod C = 0$ has a solution.

Now, assuming that we have detected a cycle using the above method, we find the start of the cycle, by first calculating the cycle length. We do this by freezing the fast pointer, and counting the number of times we have to advance the slow pointer to come back to the fast pointer. Consequently, we set both `slow` and `fast` pointers to the head. Then we advance `fast` by the length of the cycle, then move both `slow` and `fast` one at a time. The start of the cycle is located at the node where these two pointers meet again.

The code to do this traversal is quite simple in C++:

```
template <typename T>
shared_ptr<node_t<T>> has_cycle(const shared_ptr<node_t<T>> &head) {
  shared_ptr<node_t<T>> fast = head, slow = head;

  while (slow && slow->next && fast && fast->next && fast->next->next) {
    slow = slow->next, fast = fast->next->next;
    // Found cycle
    if (slow == fast) {
      // Calculate the cycle length
      int cycle_len = 0;
      do {
        ++cycle_len;
        fast = fast->next;
      } while (slow != fast);

      // Try to find the start of the cycle
      slow = head, fast = head;
      // Fast pointer advances cycle_len first
      while (cycle_len--) {
        fast = fast->next;
      }
      // Both pointers advance at the same time
```

```
      while (slow != fast) {
        slow = slow->next, fast = fast->next;
      }
      return slow;  // the start of cycle
    }
  }
  return nullptr;  // no cycle
}
```

ϵ-Variant 7.2.1: The following program purports to compute the beginning of the cycle without determining the length of the cycle; it has the benefit of being more succinct than the code listed above. Is the program correct?

```
template <typename T>
shared_ptr<node_t<T>> has_cycle(const shared_ptr<node_t<T>> &head) {
  shared_ptr<node_t<T>> fast = head, slow = head;

  while (slow && slow->next && fast && fast->next && fast->next->next) {
    slow = slow->next, fast = fast->next->next;
    // Found cycle
    if (slow == fast) {
      // Try to find the start of the cycle
      slow = head;
      // Both pointers advance at the same time
      while (slow != fast) {
        slow = slow->next, fast = fast->next;
      }
      return slow;  // slow is the start of cycle
    }
  }
  return nullptr;  // means no cycle
}
```

Problem 7.3, *pg. 63*: *Write a function that takes a sorted circular singly linked list and a pointer to an arbitrary node in this linked list, and returns the median of the linked list.*

Solution 7.3: We can solve this in stages. First we find n, the number of nodes. Then we identify the first node f with the minimum element. Finally, we return the $\lfloor \frac{n}{2} \rfloor$-th element if n is odd, with f being the 0-th element, and the average of the $\frac{n}{2}$-th and $(\frac{n}{2}+1)$-th elements if n is even. One corner case to watch out for is all entries being equal, which we check for in the first stage since we cannot find the first node with the minimum element.

```
template <typename T>
double find_median_sorted_circular_linked_list(
  const shared_ptr<node_t<T>> &r_node) {
  if (!r_node) {
    return 0.0;  // no node in this linked list
  }

  // Check all nodes are identical or not and identify the start of list
```

```
  shared_ptr<node_t<T>> curr = r_node, start = r_node;
  int count = 0;
  bool is_identical = true;
  do {
    if (curr->data != curr->next->data) {
      is_identical = false;
    }
    ++count, curr = curr->next;

    // start will point to the largest element in the list
    if (start->data <= start->next->data) {
      start = start->next;
    }
  } while (curr != r_node);
  // If all values are identical, median = curr->data
  if (is_identical == true) {
    return curr->data;
  }

  // Since start point to the largest element, its next is the start of list
  start = start->next;

  // Traverse to the middle of the list and return the median
  for (int i = 0; i < (count - 1) >> 1; ++i) {
    start = start->next;
  }
  return count & 1 ? start->data : 0.5 * (start->data + start->next->data);
}
```

Problem 7.4, *pg. 64*: *Let $h1$ and $h2$ be the heads of lists $L1$ and $L2$, respectively. Assume that $L1$ and $L2$ are well-formed, that is each consists of a finite sequence of nodes. (In particular, neither list has a cycle.) How would you determine if there exists a node r reachable from both $h1$ and $h2$ by following the next fields? If such a node exists, find the node that appears earliest when traversing the lists. You are constrained to use no more than constant additional storage.*

Solution 7.4: The lists overlap iff both have the same tail node: since each node has a single next field, once the lists converge at a node, they cannot diverge at a later node. Let $|L|$ denote the number of nodes in list L. Checking overlap amounts to finding the tail nodes for each, which is easily performed in $O(|L1| + |L2|)$ time and $O(1)$ space. To find the first node, we proceed as above, and in addition we compute $|L1|$ and $|L2|$. The first node is determined by first advancing through the longer list by $||L1| - |L2||$ nodes, and then advancing through both lists in lock-step, stopping at the first common node.

```
// Count the list length till end
template <typename T>
int count_len(shared_ptr<node_t<T>> L) {
  int len = 0;
  while (L) {
    ++len, L = L->next;
```

```
  }
  return len;
}

template <typename T>
void advance_list_by_k(shared_ptr<node_t<T>> &L, int k) {
  while (k--) {
    L = L->next;
  }
}

template <typename T>
shared_ptr<node_t<T>> overlapping_no_cycle_lists(shared_ptr<node_t<T>> L1,
                                                 shared_ptr<node_t<T>> L2) {
  // Count the lengths of L1 and L2
  int L1_len = count_len<T>(L1), L2_len = count_len<T>(L2);

  // Advance the longer list
  advance_list_by_k(L1_len > L2_len ? L1 : L2, abs(L1_len - L2_len));

  while (L1 && L2 && L1 != L2) {
    L1 = L1->next, L2 = L2->next;
  }
  return L1;  // nullptr means no overlap between L1 and L2
}
```

Figure 21.4 shows an example of lists which overlap and have cycles. For this example, both A and B are acceptable answers.

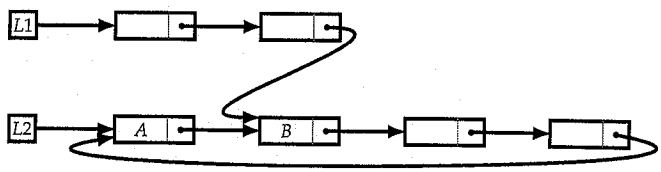


Figure 21.4: Overlapping lists.

Problem 7.5, *pg. 64*: *Solve Problem 7.4 on Page 64 for the case where L1 and L2 may each or both have a cycle. If such a node exists, return a node that appears first when traversing the lists. This node may not be unique—if L1 has a cycle $\langle n_0, n_1, \ldots, n_{k-1}, n_0 \rangle$, where n_0 is the first node encountered when traversing L1, then L2 may have the same cycle but a different first node.*

Solution 7.5: Suppose that one, or both, of the lists may have a cycle. Using the approach in Solution 7.2 on Page 208, we can determine in linear time and $O(1)$ space whether the lists have a cycle. If neither is cyclic, the lists are well-formed, and we can check overlap using the technique in Solution 7.4 on the preceding page. If one is cyclic, and the other is not, they cannot overlap.

If both are cyclic, and overlap, the cycles must be identical. Use the technique in Solution 7.2 on Page 208 to obtain nodes $a1$ and $a2$ on the cycle of $L1$ and $L2$. Visit the cycle from $a1$, stopping when $a1$ reappears. If $a2$ appears during this visit, the cycles are identical; otherwise, they are disjoint—the lists have no overlap. If there is an overlap, the problem specification allows us to return either $a1$ or $a2$. It is readily verified that the entire computation runs in $O(|L1| + |L2|)$ time and uses $O(1)$ space.

```
template <typename T>
shared_ptr<node_t<T>> overlapping_lists(shared_ptr<node_t<T>> L1,
                                        shared_ptr<node_t<T>> L2) {
  // Store the start of cycle if any
  shared_ptr<node_t<T>> s1 = has_cycle<T>(L1), s2 = has_cycle<T>(L2);

  if (!s1 && !s2) {
    return overlapping_no_cycle_lists(L1, L2);
  } else if (s1 && s2) {  // both lists have cycles
    shared_ptr<node_t<T>> temp = s2;
    do {
      temp = temp->next;
    } while (temp != s1 && temp != s2);
    return temp == s1 ? s1 : nullptr;
  }
  return nullptr;  // one list has cycle, one list has no cycle
}
```

Problem 7.6, *pg. 64*: *Write a function that takes a singly linked list L, and reorders the elements of L so that the new list represents even-odd(L). Your function should use $O(1)$ additional storage, as illustrated in Figure 7.6 on Page 65. The only field you can change in a node is* **next**.

Solution 7.6: We maintain two pointers, one iterates through the even elements, the other iterates through odd elements. We update the next field of the even pointer to the next of the odd pointer, and vice versa. Finally we update the next field of the last of the even elements to the head of the odd list. Care has to be taken to handle odd/even length lists uniformly, and to correctly process extreme cases (first and last nodes).

```
template <typename T>
shared_ptr<node_t<T>> even_odd_merge(const shared_ptr<node_t<T>> &L) {
  shared_ptr<node_t<T>> odd = L ? L->next : nullptr;
  shared_ptr<node_t<T>> odd_curr = odd;
  shared_ptr<node_t<T>> pre_even_curr = nullptr, even_curr = L;

  while (even_curr && odd_curr) {
    even_curr->next = odd_curr->next;
    pre_even_curr = even_curr;
    even_curr = even_curr->next;
    if (even_curr) {
      odd_curr->next = even_curr->next;
      odd_curr = odd_curr->next;
    }
```

```
  }

  // Odd number of nodes
  if (even_curr) {
    pre_even_curr = even_curr;
  }
  // Prevent empty list
  if (pre_even_curr) {
    pre_even_curr->next = odd;
  }
  return L;
}
```

Problem 7.7, *pg. 65*: *Let v be a node in a singly linked list L. Node v is not the tail; delete it in $O(1)$ time.*

Solution 7.7: This is more of a trick question than a conceptual one. Given the pointer to a node, it is impossible to delete it from the list without modifying its predecessor's next pointer and the only way to get to the predecessor is to traverse the list from head. However it is easy to delete the next node since it just requires modifying the next pointer of the current node. Now if we copy the value part of the next node to the current node, this would be equivalent to deleting the current node.

In practice this approach would not be acceptable, since it corrupts pointer-valued variables that point to v's successor.

```
template <typename T>
void deletion_from_list(const shared_ptr<node_t<T>> &v) {
  v->data = v->next->data;
  v->next = v->next->next;
}
```

Problem 7.8, *pg. 65*: *Given a singly linked list L and a number k, write a function to remove the k-th last element from L. Your algorithm cannot use more than a few words of storage, regardless of the length of the list. In particular, you cannot assume that it is possible to record the length of the list.*

Solution 7.8: We use two pointers, curr and ahead. First, the ahead pointer is advanced by k steps, and then curr and ahead advance in step. When ahead reaches null, curr points to the k-th last node in L, and we can remove it. Following is the code in C++:

```
template <typename T>
void remove_kth_last(shared_ptr<node_t<T>> &L, const int &k) {
  // Advance k steps first
  shared_ptr<node_t<T>> ahead = L;
  int num = k;
  while (ahead && num--) {
    ahead = ahead->next;
  }

```

```
  if (num) {
    throw length_error("not enough nodes in the list");
  }

  shared_ptr<node_t<T>> pre = nullptr, curr = L;
  // Find the k-th last node
  while (ahead) {
    pre = curr;
    curr = curr->next, ahead = ahead->next;
  }
  if (pre) {
    pre->next = curr->next;
  } else {
    L = curr->next;  // special case: delete L
  }
}
```

Problem 7.9, *pg. 65*: *Give a linear time non-recursive function that reverses a singly linked list. The function should use no more than constant storage beyond that needed for the list itself. The desired transformation is illustrated in Figure 7.7 on Page 65.*

Solution 7.9: The natural way of implementing the reversal is through recursion. However, this approach implicitly uses $\Theta(n)$ space on the stack. The function is not tail recursive, which precludes compilers from automatically converting the function to an iterative one.

Reversal can be performed iteratively—walk the list with two pointers, and update the trailing pointer's next field. It uses $O(1)$ additional storage, and has $\Theta(n)$ time complexity.

Recursive implementation, uses $\Theta(n)$ storage on the function call stack:

```
template <typename T>
shared_ptr<node_t<T>> reverse_linked_list(const shared_ptr<node_t<T>> &head) {
  if (!head || !head->next) {
    return head;
  }

  shared_ptr<node_t<T>> new_head = reverse_linked_list(head->next);
  head->next->next = head;
  head->next = nullptr;
  return new_head;
}
```

Iterative implementation:

```
template <typename T>
shared_ptr<node_t<T>> reverse_linked_list(const shared_ptr<node_t<T>> &head) {
  shared_ptr<node_t<T>> prev = nullptr, curr = head;
  while (curr) {
    shared_ptr<node_t<T>> temp = curr->next;
    curr->next = prev;
    prev = curr;
    curr = temp;
```

```
  }
  return prev;
}
```

Problem 7.10, *pg. 66*: *Write a function that determines whether a sequence represented by a singly linked list L is a palindrome. Assume L can be changed and does not have to be restored it to its original state.*

Solution 7.10: Checking if two lists represent the same sequence is straightforward. Therefore one way to check if a linked list is a palindrome is to reverse the second half of the list and compare it with the first half. The middle element can be determined by using a slow pointer and a fast pointer technique (Solution 7.2 on Page 208), and reversing a singly linked list can be performed using Solution 7.9 on the previous page.

This approach changes the list passed in, but the reversed sublist can be reversed again to restore the original list.

```
template <typename T>
bool is_linked_list_a_palindrome(shared_ptr<node_t<T>> L) {
  // Find the middle point of L
  shared_ptr<node_t<T>> slow = L, fast = L;
  while (fast) {
    fast = fast->next;
    if (fast) {
      fast = fast->next, slow = slow->next;
    }
  }

  // Compare the first half and reversed second half lists
  shared_ptr<node_t<T>> reverse = reverse_linked_list<T>(slow);
  while (reverse && L) {
    if (reverse->data != L->data) {
      return false;
    }
    reverse = reverse->next, L = L->next;
  }
  return true;
}
```

Variant 7.10.1: Solve the same problem when the list is doubly linked and you have pointers to the head and the tail.

Problem 7.11, *pg. 66*: *Write a function that takes a singly linked list L, and reorders the elements of L to form a new list representing zip(L). Your function should use $O(1)$ additional storage, as illustrated in Figure 4.1 on Page 25. The only field you can change in a node is* `next`.

Solution 7.11: The problem can be solved in a straightforward manner—find the middle of the list, reverse the second half, and then interleave the first and second

halves. The middle element can be determined by using a slow pointer and a fast pointer (Solution 7.2 on Page 208), and reversing a singly linked list can be done using Solution 7.9 on Page 215. Interleaving is performed by walking the two lists and updating next field from the first list to the corresponding element in the second list, and vice versa.

Though this algorithm is conceptually simple, corner cases abound: the empty list, lists of length 1, and even/odd lengths lists.

```
template <typename T>
void connect_a_next_to_b_advance_a(shared_ptr<node_t<T>> &a,
                                   const shared_ptr<node_t<T>> &b) {
  shared_ptr<node_t<T>> temp = a->next;
  a->next = b;
  a = temp;
}

template <typename T>
shared_ptr<node_t<T>> zipping_linked_list(const shared_ptr<node_t<T>> &L) {
  shared_ptr<node_t<T>> slow = L, fast = L, pre_slow = nullptr;

  // Find the middle point of L
  while (fast) {
    fast = fast->next;
    if (fast) {
      pre_slow = slow;
      fast = fast->next, slow = slow->next;
    }
  }

  if (!pre_slow) {
    return L;  // only contains one node in the list
  }
  pre_slow->next = nullptr;  // split the list into two lists
  shared_ptr<node_t<T>> reverse = reverse_linked_list<T>(slow), curr = L;

  // Zipping the list
  while (curr && reverse) {
    // connect curr->next to reverse, and advance curr
    connect_a_next_to_b_advance_a(curr, reverse);
    if (curr) {
      // connect reverse->next to curr, and advance reverse
      connect_a_next_to_b_advance_a(reverse, curr);
    }
  }
  return L;
}
```

Problem 7.12, *pg. 66*: *Implement a function which takes as input a pointer to the head of a postings list L, and returns a copy of the postings list. Your function should take $O(n)$ time, where n is the length of the postings list and should use $O(1)$ storage beyond that required for the n nodes in the copy. You can modify the original list, but must restore it to its initial state before returning.*

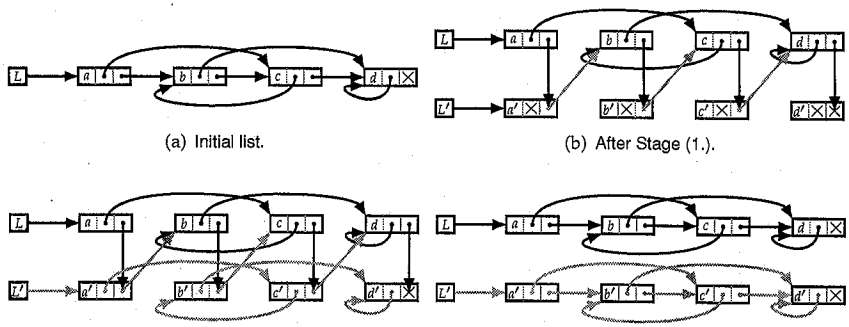
Solution 7.12: We do the copy in following three stages:

(1.) First we copy a node c_x per node x in the original list, and when we do the allocation, we set c_x's next pointer to x's next pointer, then update x's next pointer to c_x. (Note that this does not preclude us from traversing the nodes of the original list.)

(2.) Then we update the jump field for each copied node c_x; specifically, if y is x's jump field, we set c_x's jump field to c_y, which is the copied node of y. (We can do this by traversing the nodes in the original list; note that c_y is just y's next field.)

(3.) Now we set the next field for each x to its original value (which we get from c_x's next field), and the next field for each c_x to $c_{n(x)}$, where $n(x)$ is x's original next node.

These three stages are illustrated in Figures 21.5(b) to 21.5(d) on the current page.



(a) Initial list. (b) After Stage (1.).

(c) After Stage (2.). (d) After Stage (3.).

Figure 21.5: Duplicating a postings list.

Code implementing the copy is given below:

```
template <typename T>
shared_ptr<node_t<T>> copy_postings_list(const shared_ptr<node_t<T>> &L) {
  // Return empty list if L is nullptr
  if (!L) {
    return nullptr;
  }

  // 1st stage: copy the nodes from L
  shared_ptr<node_t<T>> p = L;
  while (p) {
    auto temp =
      shared_ptr<node_t<T>>(new node_t<T>{p->data, p->next, nullptr});
    p->next = temp;
    p = temp->next;
  }

  // 2nd stage: update the jump field
```

```
  p = L;
  while (p) {
    if (p->jump) {
      p->next->jump = p->jump->next;
    }
    p = p->next->next;
  }

  // 3rd stage: restore the next field
  p = L;
  shared_ptr<node_t<T>> copied = p->next;
  while (p->next) {
    shared_ptr<node_t<T>> temp = p->next;
    p->next = temp->next;
    p = temp;
  }
  return copied;
}
```

Problem 8.1, *pg. 67*: *Design a stack that supports a* **max** *operation, which returns the maximum value stored in the stack, and throws an exception if the stack is empty. Assume elements are comparable. All operations must be $O(1)$ time. You can use $O(n)$ additional space, beyond what is required for the elements themselves.*

Solution 8.1: A conceptually straightforward approach to tracking the maximum is store pairs in a stack. The first component is the key being pushed; the second is the largest value in the stack after the push is completed. When we push a value, the maximum value stored at or below any of the entries below the entry just pushed does not change. The pushed entry's maximum value is simply the larger of the value just pushed and the maximum prior to the push, which can be determined by inspecting the maximum field of the element below. Since popping does not change the values below, there is nothing special to be done for pop. Of course appropriate checks have to be made to ensure the stack is not empty.

This approach has $O(1)$ time complexity for the the specified methods. The additional space complexity is $\Theta(n)$, regardless of the stored keys.

```
template <typename T>
class Stack {
  private:
    stack<pair<T, T>> s;

  public:
    const bool empty(void) const {
      return s.empty();
    }

    const T &max(void) const {
      if (empty() == false) {
        return s.top().second;
      }
      throw length_error("empty stack");
```

```
    }

    T pop(void) {
      if (empty() == false) {
        T ret = s.top().first;
        s.pop();
        return ret;
      }
      throw length_error("empty stack");
    }

    void push(const T &x) {
      s.emplace(x, std::max(x, empty() ? x : s.top().second));
    }
};
```

Heuristically, the additional space required can be reduced by maintaining two stacks, the primary stack, which holds the keys being pushed, and an auxiliary stack, whose operation we now describe.

The top of the auxiliary stack holds a pair. The first component of the pair is the maximum key in the primary stack. The second component is the number of times that key appears in the primary stack.

Let m be the maximum key currently in the primary stack. There are three cases to consider when a key k is pushed.

1. k is smaller than m. The auxiliary stack is not updated.
2. k is equal to m. We increment the second component of the pair stored at the top of the auxiliary stack.
3. k is greater than m. The pair $(k, 1)$ is pushed onto the auxiliary stack.

There are two cases to consider when the primary stack is popped. Let k be the popped key.

1. k is less than m. The auxiliary stack is not updated.
2. k is equal to m. We decrement the second component of the top of the auxiliary stack. If its value becomes 0, we pop the auxiliary stack.

These operations are illustrated in Figure 21.6 on Page 222.

```
template <typename T>
class Stack {
  private:
    stack<T> s;
    stack<pair<T, int>> aux;

  public:
    const bool empty(void) const {
      return s.empty();
    }

    const T &max(void) const {
      if (empty() == false) {
        return aux.top().first;
      }
      throw length_error("empty stack");
```

```
    }

    T pop(void) {
      if (empty() == false) {
        T ret = s.top();
        s.pop();
        if (ret == aux.top().first) {
          --aux.top().second;
          if (aux.top().second == 0) {
            aux.pop();
          }
        }
        return ret;
      }
      throw length_error("empty stack");
    }

    void push(const T &x) {
      s.emplace(x);
      if (aux.empty() == false) {
        if (x == aux.top().first) {
          ++aux.top().second;
        } else if (x > aux.top().first) {
          aux.emplace(x, 1);
        }
      } else {
        aux.emplace(x, 1);
      }
    }
};
```

The worst-case additional space complexity is $\Theta(n)$, which occurs when each key pushed is greater than all keys in the primary stack. However, when the number of distinct keys is small, or the maximum changes infrequently, the additional space complexity is less, $O(1)$ in the best case. The time complexity for each specified method is still $O(1)$.

Problem 8.2, ***pg. 68***: *Write a function that takes an arithmetical expression in RPN and returns the number that the expression evaluates to.*

Solution 8.2: Conceptually, the algorithm for evaluating an RPN expression iterates through the string from left-to-right. It tokenizes the input into numbers and operators. Numbers are pushed onto a stack. When an operator is read, if it takes k arguments, and there are fewer than k numbers on the stack, a parse error exception is thrown. Otherwise, the expression is evaluated by popping the top k elements of the stack and the operator; the result is pushed back on the stack. When no tokens are left, the only one value left on the stack is the result; otherwise a parse error is declared.

```
int eval(const string &s) {
  stack<int> eval_stack;
  stringstream ss(s);
```

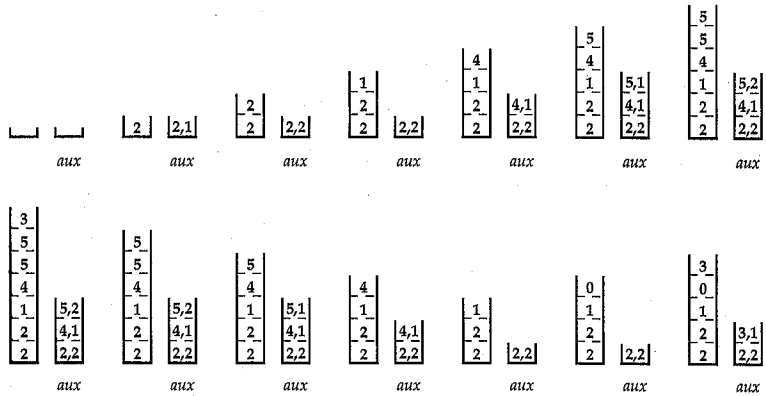


Figure 21.6: The primary and auxiliary stacks for the following operations: `push(2)`, `push(2)`, `push(1)`, `push(4)`, `push(5)`, `push(5)`, `push(3)`, `pop()`, `pop()`, `pop()`, `pop()`, `push(0)`, `push(3)`. Both stacks are initially empty, and their progression is shown from left-to-right, then top-to-bottom. The top of the auxiliary stack holds the maximum element in the stack, and the number of times that element occurs in the stack. The auxiliary stack is denoted by *aux*.

```
  string symbol;

  while (getline(ss, symbol, ',')) {
    if (symbol == "+" || symbol == "-" || symbol == "*" || symbol == "/") {
      int y = eval_stack.top();
      eval_stack.pop();
      int x = eval_stack.top();
      eval_stack.pop();
      switch (symbol.front()) {
        case '+':
          eval_stack.emplace(x + y);
          break;
        case '-':
          eval_stack.emplace(x - y);
          break;
        case '*':
          eval_stack.emplace(x * y);
          break;
        case '/':
          eval_stack.emplace(x / y);
          break;
      }
    } else {  // number
      eval_stack.emplace(stoi(symbol));
    }
  }
  return eval_stack.top();
}
```

ϵ-Variant 8.2.1: Solve the same problem for expressions in Polish notation, i.e., when

$A, B, \circ$ is replaced by $\circ, A, B$ in Rule (2.) on Page 68.

Problem 8.3, *pg. 68*: *Given a BST node n, print all the keys at n and its descendants. The nodes should be printed in sorted order, and you cannot use recursion. For example, for Node I in the binary search tree in Figure 14.1 on Page 105 you should print the sequence* $\langle 23, 29, 31, 37, 41, 43, 47, 53 \rangle$.

Solution 8.3: The recursive solution is trivial—first print the left subtree, then print the root, and finally print the right subtree. This algorithm can be converted into a iterative algorithm by using an explicit stack. Several implementations are possible; the one below is noteworthy in that it pushes the current node, and not its right child, and it does not use a visited field.

```
template <typename T>
void print_BST_in_sorted_order(const shared_ptr<BinarySearchTree<T>> &n) {
  stack<shared_ptr<BinarySearchTree<T>>> s;
  shared_ptr<BinarySearchTree<T>> curr = n;

  while (!s.empty() || curr) {
    if (curr) {
      s.push(curr);
      curr = curr->left;
    } else {
      curr = s.top();
      s.pop();
      cout << curr->data << endl;
      curr = curr->right;
    }
  }
}
```

Problem 8.4, *pg. 68*: *Write recursive and iterative routines that take a postings list, and computes the jump-first order. Assume each node has an* **order** *field, which is an integer that is initialized to* -1 *for each node.*

Solution 8.4: Recursion is natural—if the current node is unvisited, update the current node's order, visit the `jump` node, then visit the next node. The iterative solution mimics the recursive algorithm using a stack to push nodes that need to be visited. Because of a stack's last-in, first-out semantics, the `next` node is pushed first, since it is to be visited after the `jump` node. Recursive implementation:

```
template <typename T>
void search_postings_list_helper(const shared_ptr<node_t<T>> &L,
                                 int &order) {
  if (L && L->order == -1) {
    L->order = order++;
    search_postings_list_helper<T>(L->jump, order);
    search_postings_list_helper<T>(L->next, order);
  }
}

```

```
template <typename T>
void search_postings_list(const shared_ptr<node_t<T>> &L) {
  int order = 0;
  search_postings_list_helper<T>(L, order);
}
```

Iterative implementation:

```
template <typename T>
void search_postings_list(const shared_ptr<node_t<T>> &L) {
  stack<shared_ptr<node_t<T>>> s;
  int order = 0;
  s.emplace(L);
  while (!s.empty()) {
    shared_ptr<node_t<T>> curr = s.top();
    s.pop();
    if (curr && curr->order == -1) {
      curr->order = order++;
      s.emplace(curr->next);
      s.emplace(curr->jump);
    }
  }
}
```

Problem 8.5, *pg. 69*: *Exactly n rings on P1 need to be transferred to P2, possibly using P3 as an intermediate, subject to the stacking constraint. Write a function that prints a sequence of operations that transfers all the rings from P1 to P2.*

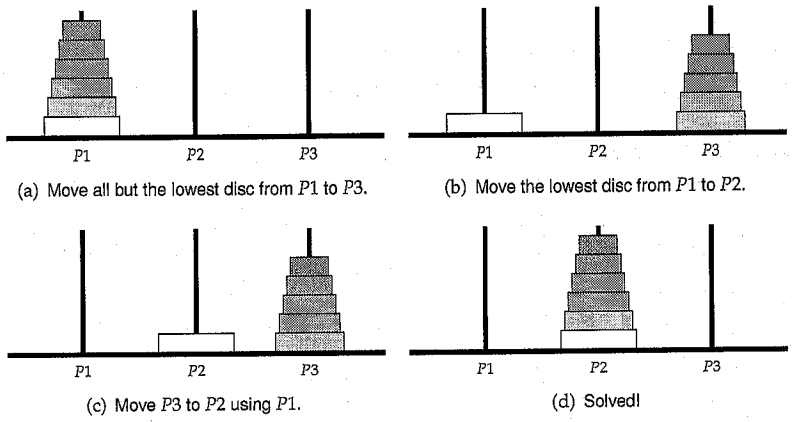
Solution 8.5: Number the n rings from 1 to n. Transfer these n rings from $P1$ to $P2$ as follows.

1. Recursively transfer $n-1$ rings from $P1$ to $P3$ using $P2$.
2. Move the ring numbered $n-1$ from $P1$ to $P2$.
3. Recursively transfer the $n-1$ rings on $P3$ to $P2$, using $P1$.

This is illustrated in Figure 21.7 on the next page. Code implementing this idea is given below.

```
void transfer(const int &n, array<stack<int>, 3> &pegs,
              const int &from, const int &to, const int &use) {
  if (n > 0) {
    transfer(n - 1, pegs, from, use, to);
    pegs[to].push(pegs[from].top());
    pegs[from].pop();
    cout << "Move from peg " << from << " to peg " << to << endl;
    transfer(n - 1, pegs, use, to, from);
  }
}

void move_tower_hanoi(const int &n) {
  array<stack<int>, 3> pegs;
  // Initialize pegs
  for (int i = n; i >= 1; --i) {
    pegs[0].push(i);
```



(a) Move all but the lowest disc from $P1$ to $P3$.

(b) Move the lowest disc from $P1$ to $P2$.

(c) Move $P3$ to $P2$ using $P1$.

(d) Solved!

Figure 21.7: A recursive solution to the Towers of Hanoi for $n = 6$.

```
  }

  transfer(n, pegs, 0, 1, 2);
}
```

ϵ-Variant 8.5.1: Find the minimum number of operations subject to the constraint that each operation must involve $P3$.

ϵ-Variant 8.5.2: Find the minimum number of operations subject to the constraint that each transfer must be from $P1$ to $P2$, $P2$ to $P3$, or $P3$ to $P1$.

ϵ-Variant 8.5.3: Find the minimum number of operations subject to the constraint that a ring can never be transferred directly from $P1$ to $P2$ (transfers from $P2$ to $P1$ are allowed).

ϵ-Variant 8.5.4: Find the minimum number of operations when the stacking constraint is relaxed to the following—the largest ring on a peg must be the lowest ring on the peg. (The remaining rings on the peg can be in any order, e.g., it is fine to have the second-largest ring above the third-largest ring.)

Variant 8.5.5: Find the minimum number of operations if you have a fourth peg, $P4$.

Problem 8.6, *pg. 69*: *Design an algorithm that processes buildings as they are presented to it and tracks the buildings that have a view of the sunset. The number of buildings is not known in advance. Buildings are given in east-to-west order and are specified by their heights. The amount of memory your algorithm uses should depend solely on the number of buildings that have a view; in particular it should not depend on the number of buildings*

processed.

Solution 8.6: We use a stack to record buildings that have a view. Each time a building b is processed, if it is taller than the building at the top of the stack, we pop the stack until the top of the stack is taller than b—all the buildings thus removed lie to the east of a taller building.

Although some individual steps may require many pops, each building is pushed and popped at most once. Therefore the run time to process n buildings is $O(n)$, and the stack always holds precisely the buildings which currently have a view.

```
template <typename T>
vector<pair<int, T>> examine_buildings_with_sunset(istringstream &sin) {
  int idx = 0; // building's index
  T height;
  // Stores (building_idx, building_height) pair with sunset views
  vector<pair<int, T>> buildings_with_sunset;
  while (sin >> height) {
    while (buildings_with_sunset.empty() == false &&
           height >= buildings_with_sunset.back().second) {
      buildings_with_sunset.pop_back();
    }
    buildings_with_sunset.emplace_back(idx++, height);
  }

  // Returns buildings with its index and height.
  return buildings_with_sunset;
}
```

ϵ-Variant 8.6.1: Solve the problem subject to the same constraints when buildings are presented in west-to-east order.

Problem 8.7, *pg. 69*: *Design an algorithm to sort a stack S of numbers in descending order. The only operations allowed are* `push`, `pop`, `top` *(which returns the top of the stack without a* `pop`*), and* `empty`*. You cannot explicitly allocate memory outside of a few words.*

Solution 8.7: We use recursion—pop the stack and store the result in e, sort the popped stack, then insert the popped element in the right place. The insertion is also done using recursion—if e is smaller than `top`, then push e and return, else do a `pop` and store the result in f, insert e in the popped stack, then push f. For both the sort and the insert functions the empty stack is the base case. This implementation uses $\Theta(n)$ storage on the function call stack:

```
template <typename T>
void insert(stack<T> &S, const T &e) {
  if (S.empty() || S.top() <= e) {
    S.push(e);
  } else {
    T f = S.top();
    S.pop();
    insert(S, e);
    S.push(f);
```

```
  }
}

template <typename T>
void sort(stack<T> &S) {
  if (!S.empty()) {
    T e = S.top();
    S.pop();
    sort(S);
    insert(S, e);
  }
}
```

Problem 8.8, *pg. 69*: *Write a function which takes a path name, and returns the shortest equivalent path name. Assume individual directories and files have names that use only alphanumeric characters. Subdirectory names may be combined using forward slashes (/), the current directory (.), and parent directory (..). The formal grammar is specified as follows:*

$$\begin{aligned} \texttt{name} &= [A-Za-z0-9]+ \\ \texttt{spdir} &= \texttt{.} \mid \texttt{..} \\ \texttt{pathname} &= \texttt{name} \mid \texttt{spdir} \mid [\texttt{spdir} \mid \texttt{name} \mid \texttt{pathname}]?/+\texttt{pathname}? \end{aligned}$$

Here + denotes one or more repetitions or the preceding token, and ? denotes 0 or 1 occurrences of the preceding token. You should throw an exception on invalid path names.

Solution 8.8: The solution uses a stack which will hold the path. The candidate string is parsed from left to right, splitting on /. A leading / is pushed on the stack—this must be an absolute path name. Consequent names are pushed on the stack. Any .. causes a pop of a nonempty stack; if the stack is empty, .. is pushed onto the stack. Any . is skipped.

The error conditions are trying to pop a stack which begins with /, and a substring which is not a name, the empty string, ., or .., separated by /.

The final state of the stack directly corresponds to the shortest equivalent directory path. The argument is based on representing the directory hierarchy as a tree rooted at the root. The directory we compute is a shortest path in the tree. If the bottom of the stack is /, the path is absolute, otherwise it is relative.

```
string normalized_path_names(const string &path) {
  vector<string> s;  // Use vector as a stack
  // Special case: starts with "/", which is an absolute path
  if (path.front() == '/') {
    s.emplace_back("/");
  }

  stringstream ss(path);
  string token;
  while (getline(ss, token, '/')) {
    if (token == "..") {
```

```
      if (s.empty() || s.back() == "..") {
        s.emplace_back(token);
      } else {
        if (s.back() == "/") {
          throw invalid_argument("Path error");
        }
        s.pop_back();
      }
    } else if (token != "." && token != "") {  // name
      for (const char &c : token) {
        if (c != '.' && isalnum(c) == false) {
          throw invalid_argument("Invalid directory name");
        }
      }
      s.emplace_back(token);
    }
  }

  string normalized_path("");
  if (s.empty() == false) {
    auto it = s.cbegin();
    normalized_path += *it++;
    while (it != s.cend()) {
      if (*(it - 1) != "/") {  // previous one is not an absolute path
        normalized_path += "/";
      }
      normalized_path += *it++;
    }
  }
  return normalized_path;
}
```

Problem 8.9, *pg. 70*: *Given the root node r of a binary tree, print all the keys and levels at r and its descendants. The nodes should be printed in order of their level. You cannot use recursion. You may use a single queue, and constant additional storage. For example, you should print the sequence* $\langle 314, 6, 6, 271, 561, 2, 271, 28, 0, 3, 1, 28, 17, 401, 257, 641 \rangle$ *for the binary tree in Figure 9.1 on Page 73.*

Solution 8.9: We maintain a queue of nodes to process. Specifically the queue contains nodes at level l followed by nodes at level $l + 1$. After all nodes from level l are processed, the head of the queue is a node at level $l + 1$; processing this node introduces nodes from level $l + 2$ to the end of the queue. We use a count variable that records the number of nodes at the level of the head of the queue that remain to be processed. When all nodes at level l are processed, the queue consists of exactly the set of nodes at level $l + 1$, and count is updated to the size of the queue.

```
template <typename T>
void print_binary_tree_level_order(const shared_ptr<BinaryTree<T>> &n) {
  // Prevent empty tree
  if (!n) {
    return;
  }
```

```

  queue<shared_ptr<BinaryTree<T>>> q;
  q.emplace(n);
  while (!q.empty()) {
    cout << q.front()->data << ' ';
    if (q.front()->left) {
      q.emplace(q.front()->left);
    }
    if (q.front()->right) {
      q.emplace(q.front()->right);
    }
    q.pop();
  }
}
```

Problem 8.10, *pg. 71*: *Implement a queue API using an array for storing elements. Your API should include a constructor function, which takes as argument the capacity of the queue,* `enqueue` *and* `dequeue` *functions, a* `size` *function, which returns the number of elements stored, and implement dynamic resizing.*

Solution 8.10: We use an array of length n to store up to n elements. We resize the array by a factor of 2 each time we run out of space. The queue has a `head` field that indexes the least recently inserted element, and a `tail` field, which is the index that the next inserted element will be written to. We record the number of elements in the queue with a `count` variable. Initially, `head` and `tail` are 0. When `count` = n and a `enqueue` is attempted we resize. When `count` = 0 and a `dequeue` is attempted we throw an exception.

```
template <typename T>
class Queue {
  private:
    size_t head, tail, count;
    vector<T> data;

  public:
    Queue(const size_t &cap = 8) : head(0), tail(0), count(0), data({cap}) {}

    void enqueue(const T &x) {
      // Dynamically resize due to data.size() limit
      if (count == data.size()) {
        data.resize(data.size() << 1);
      }
      // Perform enqueue
      data[tail] = x;
      tail = (tail + 1) % data.size(), ++count;
    }

    T dequeue(void) {
      if (count) {
        --count;
        T ret = data[head];
        head = (head + 1) % data.size();
```

```
      return ret;
    }
    throw length_error("empty queue");
  }

  const size_t &size(void) const {
    return count;
  }
};
```

Alternative implementations are possible, e.g., we can avoid using `count`, and instead use the difference between `head` and `tail` to determine the number of elements. In such an implementation we cannot store more than $n-1$ elements, since otherwise there is no way to differentiate a full queue from an empty one.

Problem 8.11, *pg. 71*: *Implement a queue using two unsigned integer-valued variables. Assume that the only elements pushed into the queue are integers in* $[0,9]$. *Your program should work correctly when* 0*s are the only elements in the queue. What is the maximum number of elements that can be stored in the queue for it to operate correctly?*

Solution 8.11: The queue state can be viewed as a sequence of digits, with the newest element corresponding to the rightmost digit. A sequence of digits uniquely represents an integer in base-10. Pushing an element corresponds to multiplying that integer by 10 and adding the new element to the result. Popping an element corresponds to identifying the most significant digit. The index i of the most significant digit d is the number of digits in the number, which is computed using $\log_{10}$; the number encoding the new queue is simply the number encoding the old queue minus $d \times 10^i$. The maximum number of elements we can store is dictated by the size of the integer. For k-bit integers the queue is limited to size $\lfloor \log_{10} 2^k \rfloor$.

```
class Queue {
  private:
    unsigned val, size;

  public:
    Queue() : val(0), size(0) {}

    void enqueue(const unsigned &x) {
      val = val * 10 + x;
      ++size;
    }

    unsigned dequeue(void) {
      if (size) {
        unsigned ret = 0, d = floor(log10(val));
        if (d + 1 == size) {
          ret = val / pow(10.0, d);
          val -= pow(10.0, d) * ret;
        }
        --size;
        return ret;
      }
```

```
        throw length_error("empty queue");
    }
};
```

ϵ-Variant 8.11.1: Implement a queue with a single integer-valued variable by reserving the most significant digit for the size.

Problem 8.12, *pg. 71*: *How would you implement a queue given two stacks and $O(1)$ additional storage? Your implementation should be efficient—the time to do a sequence of m combined enqueues and dequeues should be $O(m)$.*

Solution 8.12: Call the two stacks A and B. A straightforward implementation of the queue is to enqueue by pushing the element to be enqueued onto A. The element to be dequeued is then the element at the bottom of A, which can be achieved by first popping all the elements of A and pushing them to B, then popping the top of B (which was the bottom-most element of A), and finally popping the remaining elements from B and pushing them to A.

The primary problem with this approach is that every dequeue takes two pushes and two pops of *each* element. (Enqueue takes $O(1)$ time.)

The statement of the problem has a hint—it says that every sequence of m combined enqueues and dequeues should take $O(m)$ time. If we could implement enqueue and dequeue each on $O(1)$, this bound would be trivially met. However, the bound can also be achieved even if individual enqueues and dequeues have high time complexity, as long as there exist enough fast enqueues and dequeues to compensate.

We can implement enqueues by always pushing onto A. Dequeues are handled as follows.

- If B is empty, e.g., we have not done any dequeues so far or all elements in B have been popped, we transfer the contents of A over to B, using pops on A and pushes onto B. Now the top of B contains the element that was enqueued earliest. We simply pop and return that. We do not transfer back from B to A.
- If B is nonempty, e.g., we had just done a dequeue as above, we do by dequeuing from the top of B.

This approach takes $O(m)$ time for m operations, which can be seen from the fact that each element is pushed no more than twice (first on enqueuing onto A and then onto B) and popped no more than twice (first from A and then on dequeuing from B). This style of complexity analysis is known as amortized analysis.

Another minor observation about the implementation is that it can always hold at least n elements (since A is of size n), and in some cases, it may hold up to $2n - 1$ elements before overflowing, e.g., if we do n enqueues, a dequeue, followed by n enqueues. However, we cannot guarantee supporting more than n elements, e.g., if we do $n + 1$ consecutive enqueues, we will be forced to use stack B, and not be able to access the element enqueued earliest for a dequeue.

```
template <typename T>
```

```
class Queue {
  private:
    stack<T> A, B;

  public:
    void enqueue(const T &x) {
      A.emplace(x);
    }

    T dequeue(void) {
      if (B.empty()) {
        while (!A.empty()) {
          B.emplace(A.top());
          A.pop();
        }
      }
      if (B.empty() == false) {
        T ret = B.top();
        B.pop();
        return ret;
      }
      throw length_error("empty queue");
    }
};
```

Problem 8.13, *pg. 72*: *How would you implement a queue so that any series of m combined* enqueue, dequeue, *and* max *operations can be done in $O(m)$ time?*

Solution 8.13: This problem can be solved by a combination of Solutions 8.1 on Page 219 and 8.12 on the preceding page. Build the queue by using two stacks, each of which supports the maximum operation. This queue will be able to achieve enqueue, dequeue, and max in amortized $O(1)$ time.

```
template <typename T>
class Queue {
  private:
    Stack<T> A, B;

  public:
    void enqueue(const T &x) {
      A.push(x);
    }

    T dequeue(void) {
      if (B.empty()) {
        while (A.empty() == false) {
          B.push(A.pop());
        }
      }
      if (B.empty() == false) {
        return B.pop();
      }
      throw length_error("empty queue");
```

```
    }

    const T &max(void) const {
      if (A.empty() == false) {
        return B.empty() ? A.max() : std::max(A.max(), B.max());
      } else {  // A.empty() == true
        if (B.empty() == false) {
          return B.max();
        }
        throw length_error("empty queue");
      }
    }
};
```

The solution above is fairly indirect. A more straightforward approach is based on using a deque. Suppose the queue Q consists of elements $\langle e_0, e_1, \ldots, e_{n-1} \rangle$, where e_0 is the element at the head. Call an element e_i in Q *dominated* if there is another element e_j such that $j > i$ and $e_i < e_j$. A dominated element can never become the maximum element in Q, regardless of the sequence of enqueues and dequeues. This is because e_i will be dequeued before e_j, and $e_j > e_i$. Call e_i a *candidate* if it is not dominated.

We maintain the set of candidates in a deque D. Elements in D are ordered by their position in Q, with the candidate closest to the head of Q appearing first. Observe that each candidate in D is greater than or equal to its successors. Consequently the largest element in Q appears at the head of D.

When Q is dequeued, if the element just dequeued is at the head of D, we pop D from its head, otherwise D remains unchanged. If K is enqueued into Q, we iteratively eject D from its tail till the element at D's tail is greater than or equal to K. Then we inject K onto the tail of D. These operations are illustrated in Figure 21.8 on the next page, and implemented in the following code.

```
template <typename T>
class Queue {
  private:
    queue<T> Q;
    deque<T> D;

  public:
    void enqueue(const T &x) {
      Q.emplace(x);
      while (D.empty() == false && D.back() < x) {
        D.pop_back();
      }
      D.emplace_back(x);
    }

    T dequeue(void) {
      if (Q.empty() == false) {
        T ret = Q.front();
        if (ret == D.front()) {
          D.pop_front();
        }
```

```
        Q.pop();
        return ret;
      }
      throw length_error("empty queue");
    }

    const T &max(void) const {
      if (D.empty() == false) {
        return D.front();
      }
      throw length_error("empty queue");
    }
};
```

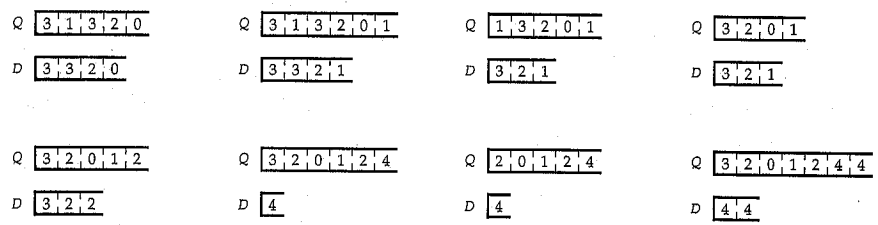


Figure 21.8: The queue with max for the following operations: `enqueue(1)`, `dequeue()`, `dequeue()`, `enqueue(2)`, `enqueue(4)`, `dequeue()`, `enqueue(4)`. The queue initially contains 3, 1, 3, 2, and 0 in that order. The deque D corresponding to queue Q is immediately below Q. The progression is shown from left-to-right, then top-to-bottom. The head of each queue and deque is on the left. Observe how the head of the deque holds the maximum element in the queue.

Each dequeue operation has time $O(1)$ complexity. A single enqueue operation may entail many ejections from D. However, the amortized time complexity of m enqueues and dequeues is $O(m)$, since an element can be added and removed from D no more than once. The max operation is $O(1)$ since it consists of returning the element at the head of D.

Problem 8.14, *pg. 72*: *Let A be an array of length n, and w the window size. Entry $A[i]$ is a pair (t_i, v_i), where t_i is the timestamp and v_i the traffic volume at that time. Assume A is sorted by increasing timestamp. Design an algorithm to compute $v_i = \max\{v_j \mid (t_i - t_j) \leq w, j \leq i\}$, for $0 \leq i \leq n-1$.*

Solution 8.14: The brute-force entails finding the maximum in the subarray consisting of elements whose timestamps line in $[A[i] - w, A[i]]$. It has time complexity $O(n\Delta_w)$, where Δ_w is the maximum number of array elements whose timestamps are contained in any length w interval.

BSTs, which are the subject of Chapter 14, can reduce the complexity to $O(n \log \Delta_w)$—one BST holds volumes in the current window, another BST implements a mapping from timestamps to corresponding nodes in the first BST.

We now describe how to use the queue with maximum data structure developed in Solution 8.13 on Page 232 to achieve an $O(n)$ time complexity, independent of w.

Initialize Q to an empty queue with maximum. Iteratively enqueue (v_i, t_i) in order of increasing i. For each i, iteratively dequeue Q until the difference of the timestamp at Q's head and t_i is less than or equal to w. The sequence of maximum values in the queue for each i is the desired result.

```
class TrafficElement {
  public:
    int time, volume;

    const bool operator<(const TrafficElement &that) const {
      return time < that.time;
    }

    const bool operator==(const TrafficElement &that) const {
      return time == that.time && volume == that.volume;
    }
};

void TrafficVolumes(const vector<TrafficElement> &A, const int &w) {
  Queue<TrafficElement> Q;
  for (int i = 0; i < A.size(); ++i) {
    Q.enqueue(A[i]);
    while (A[i].time - Q.front().time > w) {
      Q.dequeue();
    }
    cout << "Max after inserting " << i << " is " << Q.max().volume << endl;
  }
}
```

Each element is enqueued once. Each element is dequeued at most once. Since the queue with maximum data structure has an $O(1)$ amortized time complexity per operation, the overall time complexity is $O(n)$. The additional space complexity is $O(\Delta_w)$.

Problem 9.1, *pg. 75*: *Write a function that takes as input the root of a binary tree and returns* `true` *or* `false` *depending on whether the tree is balanced. Use $O(h)$ additional storage, where h is the height of the tree.*

Solution 9.1: Without the $O(h)$ constraint the problem is trivial—we can compute the height for the tree rooted at each node x recursively. The basic computation is $x.height = \max(x.left.height, x.right.height) + 1$, and in each step we check if the difference in heights of the left and right children is greater than one. We can store the heights in a hash table, or in a new field in the nodes. This entails $O(n)$ storage, where n is the number of nodes of the tree.

We will solve this problem using $O(h)$ storage by implementing a `get_height` function which takes a node x as an argument and returns an integer. The function `get_height` returns -2 if the node is unbalanced; otherwise it returns the height of the subtree rooted at that node. The implementation of `get_height` is as follows. If x is `null`, return -1. Otherwise run `get_height` on the left child. If the returned value

l is -2, node x is not balanced; return -2. Call `get_height` on x's right child; let the returned value be r. If r is -2 or $|l - r| > 1$ return -2, otherwise return $\max(l, r) + 1$.

The function `get_height` implements a postorder walk with some calls being eliminated because of early detection of unbalance. The function call stack corresponds to a sequence of calls from the root through the unique path to the current node, and the stack height is therefore bounded by the height of the tree, leading to an $O(h)$ space bound.

```
template <typename T>
int get_height(const shared_ptr<BinaryTree<T>> &n) {
  if (!n) {
    return -1;  // base case
  }

  int l_height = get_height(n->left);
  if (l_height == -2) {
    return -2;  // left subtree is not balanced
  }
  int r_height = get_height(n->right);
  if (r_height == -2) {
    return -2;  // right subtree is not balanced
  }

  if (abs(l_height - r_height) > 1) {
    return -2;  // current node n is not balanced
  }
  return max(l_height, r_height) + 1;  // return the height
}

template <typename T>
bool is_balanced_binary_tree(const shared_ptr<BinaryTree<T>> &n) {
  return get_height(n) != -2;
}
```

We can improve the space complexity if we know the number of nodes n in the tree in advance. Specifically, the space complexity can be improved to $O(\log n)$ by keeping a global variable that records the maximum height m_s of the stack. Donald Knuth (*"The Art of Computer Programming, Volume 3: Sorting and Searching"*, Page 460) proves that the height of a balanced tree on n nodes is no more than $h_n = 1.4405 \lg(\frac{n}{2} + 3) - 0.3277$. The stack height is a lower bound on the height of the tree, and therefore if the stack height ever exceeds h_n, we return -2.

Variant 9.1.1: Write a function that returns the size of the largest subtree that is complete.

Problem 9.2, *pg. 75*: *Design an algorithm that takes as input a binary tree and positive integer k, and returns a node u in the binary tree such that u is not k-balanced, but all of u's descendants are k-balanced. If no such node exists, return* `null`. *For example, when applied to the binary tree in Figure 9.1 on Page 73, your algorithm should return Node J if* $k = 3$.

Solution 9.2: It is straightforward to compute the number of nodes in each subtree of a binary tree by a postorder traversal: for each node u, count the number of nodes in its left and the right subtrees, and add one to the sum of those counts to get the count for u.

We can extend this computation by keeping a global pointer-valued variable g that is used to record the result. Initially, g is `null`. We do the postorder traversal to compute the number of nodes in subtrees as before, as long as g is not `null`. The first time our traversal finds a node which is not k-balanced, we set g to r. In the traversal, if g is not `null`, we return. The global g holds the final result.

Since it is poor programming practice to use global variables, our implementation below uses a pointer-integer pair for the return value. The pointer plays the role of g as described above. Since each node is processed only after its descendants, we are guaranteed that the result is set correctly. The time complexity is $O(n)$, where n is the number of nodes.

```
template <typename T>
pair<shared_ptr<BinaryTree<T>>, int> find_non_k_balanced_node_helper(
    const shared_ptr<BinaryTree<T>> &n, const int &k) {
  // Empty tree
  if (!n) {
    return {nullptr, 0};
  }

  // Early return if left subtree is not k-balanced
  auto L = find_non_k_balanced_node_helper<T>(n->left, k);
  if (L.first) {
    return L;
  }
  // Early return if right subtree is not k-balanced
  auto R = find_non_k_balanced_node_helper<T>(n->right, k);
  if (R.first) {
    return R;
  }

  int node_num = L.second + R.second + 1;  // #nodes in n
  if (abs(L.second - R.second) > k) {
    return {n, node_num};
  }
  return {nullptr, node_num};
}

template <typename T>
shared_ptr<BinaryTree<T>> find_non_k_balanced_node(
    const shared_ptr<BinaryTree<T>> &n, const int &k) {
  return find_non_k_balanced_node_helper<T>(n, k).first;
}
```

Problem 9.3, ***pg. 76:*** *Write a function that takes as input the root of a binary tree and returns* `true` *or* `false` *depending on whether the tree is symmetric.*

Solution 9.3: We present a recursive algorithm that follows directly from the definition of symmetry.

```
template <typename T>
bool is_symmetric_helper(const shared_ptr<BinaryTree<T>> &l,
                         const shared_ptr<BinaryTree<T>> &r) {
  if (!l && !r) {
    return true;
  } else if (l && r) {
    return l->data == r->data && is_symmetric_helper<T>(l->left, r->right) &&
           is_symmetric_helper<T>(l->right, r->left);
  } else {  // (l && !r) || (!l && r)
    return false;
  }
}

template <typename T>
bool is_symmetric(const shared_ptr<BinaryTree<T>> &n) {
  return (!n || is_symmetric_helper<T>(n->left, n->right));
}
```

Problem 9.4, *pg. 76*: *For a certain application, processes need to lock nodes in a binary tree. Implement a library for locking nodes in a binary tree, subject to the constraint that a node cannot be locked if any of its descendants or ancestors are locked. Specifically, write functions* `isLock()`, `lock()`, *and* `unLock()`, *with time complexities* $O(1)$, $O(h)$, *and* $O(h)$. *Here* h *is the height of the binary tree. Assume that each node has a parent field.*

Solution 9.4: Each node has a bool-valued `locked` field, indicating whether it holds a lock—this makes the `isLock()` function trivial. In addition, we use an integer-valued `numChildrenLocks` field for each node n which tracks the number of children in the subtrees rooted at n that are locked. The `lock()` function proceeds only if the number of locked children is 0; if so it checks the state of all the ancestors leading up to the root. If the node is lockable, the function increments the `numChildrenLocks` fields for each of the ancestors. The `unLock()` function simply sets `locked` to `false`, and decreases the `numChildrenLocks` field for each ancestor all the way to the root. The time complexity for `lock()` and `unLock()` is bounded by the distance of the node from the root, i.e., the height of the tree.

```
template <typename T>
class BinaryTree {
  private:
    bool locked;
    int numChildrenLocks;

  public:
    shared_ptr<BinaryTree<T>> left, right, parent;

    const bool &isLock(void) const {
      return locked;
    }

```

```
    void lock(void) {
      if (numChildrenLocks == 0 && locked == false) {
        // Make sure all parents do not lock
        shared_ptr<BinaryTree<T>> n = parent;
        while (n) {
          if (n->locked == true) {
            return;
          }
          n = n->parent;
        }

        // Lock itself and update its parents
        locked = true;
        n = parent;
        while (n) {
          ++n->numChildrenLocks;
          n = n->parent;
        }
      }
    }

    void unLock(void) {
      if (locked) {
        // Unlock itself and update its parents
        locked = false;
        shared_ptr<BinaryTree<T>> n = parent;
        while (n) {
          --n->numChildrenLocks;
          n = n->parent;
        }
      }
    }
};
```

Problem 9.5, *pg. 76*: *Let T be the root of a binary tree in which nodes have an explicit parent field. Design an iterative algorithm that enumerates the nodes inorder and uses $O(1)$ additional space. Your algorithm cannot modify the tree.*

Solution 9.5: The standard idiom for an inorder walk is visit-left, visit-root, visit-right. Accessing the left child is straightforward. Returning from a left child l to its parent entails examining l's parent field; returning from a right child r to its parent is similar.

To make this scheme work, we need to know when we take a parent pointer to node `r` if the child we completed visiting was `r`'s left child (in which case we need to visit `r` and then `r`'s right child) or a right child (in which case we have completed visiting `r`). We achieve this by storing the child in a `prev` variable before we move to the parent, `r`. We then compare `prev` with `r`'s left child and the right child.

```
template <typename T>
void inorder_traversal(const shared_ptr<BinaryTree<T>> &r) {
  // Empty tree
  if (!r) {
```

```
    return;
  }

  shared_ptr<BinaryTree<T>> prev = nullptr, curr = r, next;
  while (curr) {
    if (!prev || prev->left == curr || prev->right == curr) {
      if (curr->left) {
        next = curr->left;
      } else {
        cout << curr->data << endl;
        next = (curr->right ? curr->right : curr->parent);
      }
    } else if (curr->left == prev) {
      cout << curr->data << endl;
      next = (curr->right ? curr->right : curr->parent);
    } else {  // curr->right == prev
      next = curr->parent;
    }

    prev = curr;
    curr = next;
  }
}
```

ϵ-Variant 9.5.1: How would you perform preorder and postorder walks iteratively using $O(1)$ additional space? Your algorithm cannot modify the tree. Nodes have an explicit parent field.

Problem 9.6, *pg. 76*: *Design a function that efficiently computes the k-th node appearing in an inorder traversal. Specifically, your function should take as input a binary tree T and an integer k. Each node has a* `size` *field, which is the number of nodes in the subtree rooted at that node. What is the time complexity of your function?*

Solution 9.6: If the left child has $k-1$ children, then the root is the k-th node; if the left child has k or more children, then the k-th node is the k-th node of the left subtree; and if the left child has $l < k-1$ children, the k-th node is the $k-(l+1)$-th node of the right subtree.

```
template <typename T>
shared_ptr<BinaryTree<T>> find_kth_node_binary_tree(
    shared_ptr<BinaryTree<T>> r, int k) {
  while (k && r) {
    int left_size = r->left ? r->left->size : 0;
    if (left_size < k - 1) {
      k -= (left_size + 1);
      r = r->right;
    } else if (left_size == k - 1) {
      return r;
    } else {  // left_size > k - 1
      r = r->left;
    }
  }
```

```
  throw length_error("no k-th node in binary tree");
}
```

Since we descend the tree in each iteration, the time complexity is $O(h)$, where h is the height of the tree.

Problem 9.7, *pg. 77*: *Given an inorder traversal order, and one of a preorder or a postorder traversal order of a binary tree, write a function to reconstruct the tree.*

Solution 9.7: Suppose we are given the inorder and preorder traversal sequences. The preorder sequence gives us the key of the root node—it is the first node in the sequence. This in turn allows us to split the inorder sequence into an inorder sequence for the left subtree, followed by the root, followed by the right subtree. The left subtree inorder sequence allows us to compute the preorder sequence for the left subtree from the preorder sequence: the nodes in the left subtree appear before all the nodes in the right subtree in the preorder sequence.

For example, if the inorder sequence is $\langle B, A, D, C, E\rangle$ and the preorder sequence is $\langle A, B, C, D, E\rangle$, we know the root has key A and the left subtree consists of the single node with key B. Therefore, in an inorder visit the right subtree nodes appear as $\langle D, C, E\rangle$, and in a preorder visit the right subtree nodes appear as $\langle C, D, E\rangle$. Therefore the root of the right subtree is a node whose key is C and its left and right subtrees are the single nodes with keys D and E, respectively.

If the keys are unique, we can use the above algorithm to uniquely reconstruct a binary tree yielding the given inorder and preorder sequences, but this is not always true if duplicate keys are present. As an extreme example, if all keys are the same, all binary trees on n nodes with that key yield identical sequences.

```
template <typename T>
shared_ptr<BinaryTree<T>> reconstruct_pre_in_orders_helper(
    const vector<T> &pre, const int &pre_s, const int &pre_e,
    const vector<T> &in, const int &in_s, const int &in_e) {
  if (pre_e > pre_s && in_e > in_s) {
    auto it = find(in.cbegin() + in_s, in.cbegin() + in_e, pre[pre_s]);
    int left_tree_size = it - (in.cbegin() + in_s);

    return shared_ptr<BinaryTree<T>>(new BinaryTree<T>{
      pre[pre_s],
      // Recursively build the left subtree
      reconstruct_pre_in_orders_helper<T>(
        pre, pre_s + 1, pre_s + 1 + left_tree_size,
        in, in_s, it - in.cbegin()),
      // Recursively build the right subtree
      reconstruct_pre_in_orders_helper<T>(
        pre, pre_s + 1 + left_tree_size, pre_e,
        in, it - in.cbegin() + 1, in_e)
      });
  }
  return nullptr;
}

```

```
template <typename T>
shared_ptr<BinaryTree<T>> reconstruct_pre_in_orders(const vector<T> &pre,
                                                    const vector<T> &in) {
  return reconstruct_pre_in_orders_helper(pre, 0, pre.size(),
                                          in, 0, in.size());
}
```

We recover the tree from postorder and inorder traversal sequences similarly:

```
template <typename T>
shared_ptr<BinaryTree<T>> reconstruct_post_in_orders_helper(
    const vector<T> &post, const int &post_s, const int &post_e,
    const vector<T> &in, const int &in_s, const int &in_e) {
  if (post_e > post_s && in_e > in_s) {
    auto it = find(in.cbegin() + in_s, in.cbegin() + in_e, post[post_e - 1]);
    int left_tree_size = it - (in.cbegin() + in_s);

    return shared_ptr<BinaryTree<T>>(new BinaryTree<T>{
      post[post_e - 1],
      // Recursively build the left subtree
      reconstruct_post_in_orders_helper<T>(
        post, post_s, post_s + left_tree_size,
        in, in_s, it - in.cbegin()),
      // Recursively build the right subtree
      reconstruct_post_in_orders_helper<T>(
        post, post_s + left_tree_size, post_e - 1,
        in, it - in.cbegin() + 1, in_e)
      });
  }
  return nullptr;
}

template <typename T>
shared_ptr<BinaryTree<T>> reconstruct_post_in_orders(const vector<T> &post,
                                                     const vector<T> &in) {
  return reconstruct_post_in_orders_helper(post, 0, post.size(),
                                           in, 0, in.size());
}
```

Variant 9.7.1: Let A be an array of n distinct integers. Let the index of the maximum element of A be m. Define the max-tree on A to be the binary tree on the entries of A in which the root contains the maximum element of A, the left child is the max-tree on $A[0:m-1]$ and the right child is the max-tree on $A[m+1:n-1]$. Design an $O(n)$ algorithm for building the max-tree of A.

Problem 9.8, *pg. 77*: *Design an $O(n)$ time algorithm for reconstructing a binary tree from a preorder visit sequence that uses* `null` *to mark empty children. How would you modify your reconstruction algorithm if the sequence corresponded to a postorder or inorder walk?*

Solution 9.8: We traverse the sequence from right-to-left. We push nodes and `null`s on to a stack; every time we encountered a non-`null` node x, we pop the stack twice—call the first node popped `l` and the second `r`. Set x's left and right children to `l` and

r, respectively, and push x. When the sequence is exhausted, there will be a single node on the stack, which will be the root.

```
template <typename T>
shared_ptr<BinaryTree<T>> reconstruct_preorder(
  const vector<shared_ptr<T>> &preorder) {
  stack<shared_ptr<BinaryTree<T>>> s;
  for (auto it = preorder.crbegin(); it != preorder.crend(); ++it) {
    if (!(*it)) {
      s.emplace(nullptr);
    } else {  // non-nullptr
      shared_ptr<BinaryTree<T>> l = s.top();
      s.pop();
      shared_ptr<BinaryTree<T>> r = s.top();
      s.pop();
      s.emplace(new BinaryTree<T>{*(*it), l, r});
    }
  }
  return s.top();
}
```

Reconstructing from a postorder traversal is similar—we traverse from the beginning of the sequence, and when popping, the top of the stack is the right child, and the node below it is the left child.

Reconstructing from an inorder traversal is impossible, even with the null markers. This is because *every* binary tree that yields $\langle v_0, v_1, \ldots, v_{n-1} \rangle$ on an inorder walk has a modified sequence of $\langle \texttt{null}, v_0, \texttt{null}, v_1, \texttt{null}, v_2, \ldots, \texttt{null}, v_{n-1}, \texttt{null} \rangle$. An inorder traversal order is not enough to uniquely reconstruct a binary tree, so the inorder sequence with markers will also be insufficient. If all we want is a binary tree that yields the given sequence, we can simply return a completely right-skewed tree, i.e., its root is v_0, left child is empty, and right child is reconstructed recursively from $\langle v_1, v_2, \ldots, v_{n-1} \rangle$.

Problem 9.9, *pg. 78*: *Given a binary tree, write a function which forms a linked list from the leaves of the binary tree. The leaves should appear in left-to-right order. For example, when applied to the binary tree in Figure 9.1 on Page 73, your function should return* $\langle D, E, H, M, N, P \rangle$.

Solution 9.9: We use recursion, passing in the list L of leaves. If the node is a leaf, which we determine by checking if both children are null, we append it to the list L and return. Otherwise, we recurse on the left and right children, which causes the leaves on the left subtree to appear before the leaves on the right subtree. The time complexity is $O(n)$, where n is the number of nodes.

```
template <typename T>
void connect_leaves_helper(const shared_ptr<BinaryTree<T>> &n,
                           list<shared_ptr<BinaryTree<T>>> &L) {
  if (n) {
    if (!n->left && !n->right) {
      L.push_back(n);
    } else {
```

```
      connect_leaves_helper(n->left, L);
      connect_leaves_helper(n->right, L);
    }
  }
}

template <typename T>
list<shared_ptr<BinaryTree<T>>> connect_leaves(
    const shared_ptr<BinaryTree<T>> &n) {
  list<shared_ptr<BinaryTree<T>>> L;
  connect_leaves_helper(n, L);
  return L;
}
```

Problem 9.10, *pg. 78*: *Write a function that prints the nodes on the exterior of a binary tree in anti-clockwise order, i.e., print the nodes on the path from the root to the leftmost leaf in that order, then the leaves from left-to-right, then the nodes from the rightmost leaf up to the root. For example, when applied to the binary tree in Figure 9.1 on Page 73, your function should return* $\langle A, B, C, D, E, H, M, N, P, O, I \rangle$. *(By leftmost (rightmost) leaf, we mean the leaf that appears first (last) in an inorder walk.)*

Solution 9.10: One approach is to print all the nodes leading to the leftmost leaf first, using a recursive search from the root that favors a left child when available, followed by the leaves (which can be performed using the technique in Solution 9.9 on the preceding page) followed by printing all nodes from the rightmost leaf to the root, which is performed using a recursive search from the root that favors the right. The first and last functions print in preorder and postorder to ensure the right ordering of nodes.

Alternately, we can print the root, followed by all the required nodes (leftmost and leaves) from the left subtree followed by all the required nodes (leaves and rightmost) from the right subtree. The left subtree and right subtree are processed by symmetric functions. Details are given below.

```
template <typename T>
void left_boundary_b_tree(const shared_ptr<BinaryTree<T>> &n,
                          const bool &is_boundary) {
  if (n) {
    if (is_boundary || (!n->left && !n->right)) {
      cout << n->data << ' ';
    }
    left_boundary_b_tree(n->left, is_boundary);
    left_boundary_b_tree(n->right, is_boundary && !n->left);
  }
}

template <typename T>
void right_boundary_b_tree(const shared_ptr<BinaryTree<T>> &n,
                           const bool &is_boundary) {
  if (n) {
    right_boundary_b_tree(n->left, is_boundary && !n->right);
    right_boundary_b_tree(n->right, is_boundary);
```

```
    if (is_boundary || (!n->left && !n->right)) {
      cout << n->data << ' ';
    }
  }
}

template <typename T>
void exterior_binary_tree(const shared_ptr<BinaryTree<T>> &root) {
  if (root) {
    cout << root->data << ' ';
    left_boundary_b_tree(root->left, true);
    right_boundary_b_tree(root->right, true);
  }
}
```

Problem 9.11, *pg. 78*: *Design an efficient algorithm for computing the LCA of nodes a and b in a binary tree in which nodes do not have a parent pointer.*

Solution 9.11: Let a and b be the nodes whose LCA we wish to compute. Observe that if the root is one of a or b, then it is the LCA. Otherwise, let L and R be the trees rooted at the left child and the right child of the root. If both nodes lie in L (or R), their LCA is in L (or R). Otherwise, their LCA is the root itself. This is the basis for the algorithm presented below. Its time complexity is $O(n)$, where n is the number of nodes.

```
template <typename T>
shared_ptr<BinaryTree<T>> LCA(const shared_ptr<BinaryTree<T>> &n,
                              const shared_ptr<BinaryTree<T>> &a,
                              const shared_ptr<BinaryTree<T>> &b) {
  if (!n) {  // empty subtree
    return nullptr;
  } else if (n == a || n == b) {
    return n;
  }

  auto l_res = LCA(n->left, a, b), r_res = LCA(n->right, a, b);
  if (l_res && r_res) {
    return n;  // found a and b in different subtrees
  } else {
    return l_res ? l_res : r_res;
  }
}
```

Problem 9.12, *pg. 78*: *Given two nodes in a binary tree T, design an algorithm that computes their LCA. Assume that each node has a parent pointer. The tree has n nodes and height h. Your algorithm should run in $O(1)$ space and $O(h)$ time.*

Solution 9.12: Suppose we know the depths d_a and d_b of nodes a and b. Without loss of generality, assume $d_a \geq d_b$. Follow $d_a - d_b$ parent pointers starting from a. Let c be the resulting node. The LCA of a and b is the same as the LCA of b and c. We can

now compute the LCA of c and b by iteratively moving up the tree, from c and from b till we reach a common node l, which is the desired LCA.

The depth of a node can be computed by following its parent pointers until the root is reached. This computation has a time complexity $O(h)$, and space complexity $O(1)$. Therefore the LCA of a and b can be computed in $O(h)$ time and $O(1)$ space.

```
template <typename T>
int get_depth(shared_ptr<BinaryTree<T>> n) {
  int d = 0;
  while (n) {
    ++d, n = n->parent;
  }
  return d;
}

template <typename T>
shared_ptr<BinaryTree<T>> LCA(shared_ptr<BinaryTree<T>> a,
                              shared_ptr<BinaryTree<T>> b) {
  int depth_a = get_depth(a), depth_b = get_depth(b);
  if (depth_b > depth_a) {
    swap(a, b);
  }

  // Advance deeper node first
  int depth_diff = depth_a - depth_b;
  while (depth_diff--) {
    a = a->parent;
  }

  // Both pointers advance until they found a common ancestor
  while (a != b) {
    a = a->parent, b = b->parent;
  }
  return a;
}
```

Problem 9.13, *pg. 78*: *Design an algorithm for computing the LCA of a and b that has time complexity* $O(\max(d_a - d_l, d_b - d_l))$. *What is the worst-case time and space complexity of your algorithm?*

Solution 9.13: Let the sequences of nodes as we traverse parent pointers from a and b to the root be $\langle a, a_1, a_2, \ldots \rangle$ and $\langle b, b_1, b_2, \ldots \rangle$ respectively. The LCA of a and b is the first node in either sequence that is common to the two sequences. This leads to an algorithm for computing the LCA in $O(\max(d_a - d_l, d_b - d_l))$ time: interleave traversing parent pointers from a and from b, storing visited nodes in a hash table. Each time we visit a node we check to see if it has been visited before. We will revisit a node after exactly $2(\max(d_a, d_b) - d_l)$ pointers have been traversed, yielding the desired time complexity.

Note that we are trading space for time. The algorithm for Solution 9.12 on the previous page used $O(1)$ space and $O(h)$ time, whereas the algorithm presented above

uses $O(\max(d_a, d_b) - d_l)$ space and time. In the worst case, a and b are leaves whose LCA is the root, and we use $O(h)$ space and time.

```
template <typename T>
shared_ptr<BinaryTree<T>> LCA(shared_ptr<BinaryTree<T>> a,
                              shared_ptr<BinaryTree<T>> b) {
  unordered_set<shared_ptr<BinaryTree<T>>> hash;
  while (a || b) {
    if (a) {
      if (hash.emplace(a).second == false) {
        return a;  // adds a failed because a exists in hash
      }
      a = a->parent;
    }
    if (b) {
      if (hash.emplace(b).second == false) {
        return b;  // adds b failed because b exists in hash
      }
      b = b->parent;
    }
  }
  // Throw error if a and b are not in the same tree
  throw invalid_argument("a and b are not in the same tree");
}
```

Problem 9.14, *pg. 79*: *Given a string s and a set of strings D, find the shortest prefix of s which is not a prefix of any string in D.*

Solution 9.14: A trie is a data structure for storing a set of strings based on positional trees. To be concrete, suppose the strings are over the alphabet {"a", "b", ..., "z"}. Each node has a hash table mapping each character in the alphabet to the corresponding child pointer. Some or all of the children may be `null`. A path of length l starting from the root naturally corresponds to a string of l characters. Each node has a Boolean field indicating whether the string corresponding to the path from the root is a string in the set.

Finding a shortest prefix of s that is not a prefix of any string in the represented set is simply a matter of finding the first node m on the search path from the root that does not have a child corresponding to the next character in s.

```
class Trie {
  private:
    class TrieNode {
      public:
        bool isString;
        unordered_map<char, shared_ptr<TrieNode>> l;
    };

    shared_ptr<TrieNode> root;

  public:
    Trie() : root(shared_ptr<TrieNode>(new TrieNode{false})) {}

```

```
    bool insert(const string &s) {
      shared_ptr<TrieNode> p = root;
      for (const char &c : s) {
        if (p->l.find(c) == p->l.cend()) {
          p->l[c] = shared_ptr<TrieNode>(new TrieNode{false});
        }
        p = p->l[c];
      }

      // s already existed in this trie
      if (p->isString == true) {
        return false;
      } else {  // p->isString == false
        p->isString = true;  // inserts s into this trie
        return true;
      }
    }

    string getShortestUniquePrefix(const string &s) {
      shared_ptr<TrieNode> p = root;
      string prefix;
      for (const char &c : s) {
        prefix += c;
        if (p->l.find(c) == p->l.cend()) {
          return prefix;
        }
        p = p->l[c];
      }
      return {};
    }
};

string find_shortest_prefix(const string &s, const unordered_set<string> &D) {
  // Build a trie according to given dictionary D
  Trie T;
  for (const string &word : D) {
    T.insert(word);
  }
  return T.getShortestUniquePrefix(s);
}
```

e-**Variant 9.14.1**: How would you find the shortest string that is not a prefix of any string in D?

Problem 10.1, ***pg. 80***: *Design an algorithm that takes a set of files containing stock trade information in sorted order, and writes a single file containing the lines appearing in the individual files sorted in sorted order. The algorithm should use very little RAM, ideally of the order of a few kilobytes.*

Solution 10.1: In the abstract, we are trying to merge k sorted files. One way to do this is to repeatedly pick the smallest element amongst the smallest remaining elements from each file. A min-heap is ideal for maintaining a set of elements when

we repeatedly insert and query for the smallest element (both extract-min and insert take $O(\log k)$ time). Hence we can do the merge in $O(n \log k)$ time, where n is the total number of elements in the input. Here is the code for this:

```
template <typename T>
class Compare {
  public:
    const bool operator()(const pair<T, int> &lhs,
                          const pair<T, int> &rhs) const {
      return lhs.first > rhs.first;
    }
};

template <typename T>
vector<T> merge_arrays(const vector<vector<T>> &S) {
  priority_queue<pair<T, int>, vector<pair<T, int>>, Compare<T>> min_heap;
  vector<int> S_idx(S.size(), 0);

  // Every array in S puts its smallest element in heap
  for (int i = 0; i < S.size(); ++i) {
    if (S[i].size() > 0) {
      min_heap.emplace(S[i][0], i);
      S_idx[i] = 1;
    }
  }

  vector<T> ret;
  while (!min_heap.empty()) {
    pair<T, int> p = min_heap.top();
    ret.emplace_back(p.first);
    // Add the smallest element into heap if possible
    if (S_idx[p.second] < S[p.second].size()) {
      min_heap.emplace(S[p.second][S_idx[p.second]++], p.second);
    }
    min_heap.pop();
  }
  return ret;
}
```

Alternately, we could recursively merge the k files, two at a time using the merge step from merge sort.

Problem 10.2, *pg. 81*: *Design an efficient algorithm for sorting a k-increasing-decreasing array. You are given another array of the same size that the result should be written to, and you can use $O(k)$ additional storage.*

Solution 10.2: The first thing to note is that any array can be decomposed into a sequence of increasing and decreasing subarrays. If k is comparable to n, then the problem is equivalent to the general sorting problem.

If k is substantially smaller than n, we could first reverse the order of the decreasing subarrays. Now we can use the techniques in Solution 10.1 on the facing page to sort the array in time $O(n \log k)$ time with $O(k)$ space.

```
template <typename T>
vector<T> sort_k_increasing_decreasing_array(const vector<T> &A) {
  // Decompose A into a set of sorted arrays
  vector<vector<T>> S;
  bool is_increasing = true;  // the trend we are looking for
  int start_idx = 0;
  for (int i = 1; i < A.size(); ++i) {
    if ((A[i - 1] < A[i] && !is_increasing) ||
        (A[i - 1] >= A[i] && is_increasing)) {
      if (is_increasing) {
        S.emplace_back(A.cbegin() + start_idx, A.cbegin() + i);
      } else {
        S.emplace_back(A.crbegin() + A.size() - i,
                       A.crbegin() + A.size() - start_idx);
      }
      start_idx = i;
      is_increasing = !is_increasing;  // inverse the trend we are looking for
    }
  }
  if (start_idx < A.size()) {
    if (is_increasing) {
      S.emplace_back(A.cbegin() + start_idx, A.cend());
    } else {
      S.emplace_back(A.crbegin(), A.crbegin() + A.size() - start_idx);
    }
  }

  return merge_arrays(S);
}
```

Problem 10.3, *pg. 81*: *How would you implement a stack API using a heap and a queue API using a heap?*

Solution 10.3: The basic idea is to use an integer-valued variable order that keeps track of the order in which elements were added.

We mimic a stack S with a max-heap H by storing $y = (\text{order}, x)$ in H each time x is pushed in S, and incrementing order. Heap entries are compared by order. Popping is simply a matter of extracting the max element.

We mimic a queue analogously, except that we decrement order on inserts, thereby favoring the element that was inserted first when we do extract-max. It is straightforward to support queue inserts and deletes. Supporting a back function, which returns the element at the queue tail, is more involved. It can be performed with an additional min-heap.

```
template <typename T>
class Compare {
  public:
    bool operator()(const pair<int, T> &lhs, const pair<int, T> &rhs) const {
      return lhs.first < rhs.first;
    }
};
```

```

template <typename T>
class Stack :  // inherits empty(), pop(), and size() methods
  public priority_queue<pair<int, T>, vector<pair<int, T>>, Compare<T>> {
  private:
    int order;
    typedef priority_queue<pair<int, T>, vector<pair<int, T>>, Compare<T>> PQ;

  public:
    Stack() : order(0) {}

    const T &top() const {
      return PQ::top().second;
    }

    void push(const T &x) {
      PQ::emplace(order++, x);
    }
};

template <typename T>
class Queue :  // inherits empty(), pop(), and size() methods
  public priority_queue<pair<int, T>, vector<pair<int, T>>, Compare<T>> {
  private:
    int order;
    typedef priority_queue<pair<int, T>, vector<pair<int, T>>, Compare<T>> PQ;

  public:
    Queue() : order(0) {}

    const T &front() const {
      return PQ::top().second;
    }

    void push(const T &x) {
      PQ::emplace(order--, x);
    }
};
```

Problem 10.4, *pg. 81*: *How would you compute the k stars which are closest to the Earth? You have only a few megabytes of RAM.*

Solution 10.4: If RAM was not a limitation, we could read the data into an array, and apply the selection algorithm from Solution 11.13 on Page 270.

It is not difficult to come up with an algorithm based on processing through the file, selecting all stars within a distance d, and sorting the result. Selecting d appropriately is difficult, and will require multiple passes with different choices of d.

A better approach is to use a max-heap H of k elements. We start by adding the first k stars to H. As we process the stars, each time we encounter a star s that is closer to the Earth than the star m in H that is furthest from the Earth (which is the star at the root of H), we delete m from H, and add s to H.

The heap-based algorithm has $O(n \log k)$ time complexity to find the k closest stars out of n candidates, independent of the order in which stars are processed and their locations. Its space complexity is $O(k)$.

```
class Star {
  public:
    int ID;
    double x, y, z;

    // The distance between this star to the Earth
    const double distance() const {
      return sqrt(x * x + y * y + z * z);
    }

    const bool operator<(const Star &s) const {
      return distance() < s.distance();
    }
};

vector<Star> find_closest_k_stars(istringstream &sin, const int &k) {
  // Use max_heap to find the closest k stars
  priority_queue<Star, vector<Star>> max_heap;
  string line;

  // Record the first k stars
  while (getline(sin, line)) {
    stringstream line_stream(line);
    string buf;
    getline(line_stream, buf, ',');
    int ID = stoi(buf);
    array<double, 3> data;  // stores x, y, and z
    for (int i = 0; i < 3; ++i) {
      getline(line_stream, buf, ',');
      data[i] = stod(buf);
    }
    Star s{ID, data[0], data[1], data[2]};

    if (max_heap.size() == k) {
      // Compare the top of heap with the incoming star
      Star far_star = max_heap.top();
      if (s < far_star) {
        max_heap.pop();
        max_heap.emplace(s);
      }
    } else {
      max_heap.emplace(s);
    }
  }

  // Store the closest k stars
  vector<Star> closest_stars;
  while (!max_heap.empty()) {
    closest_stars.emplace_back(max_heap.top());
    max_heap.pop();
  }
```

```
  return closest_stars;
}
```

Problem 10.5, *pg. 81*: *Design an $O(n \log k)$ time algorithm that reads a sequence of n elements and for each element, starting from the k-th element, prints the k-th largest element read up to that point. The length of the sequence is not known in advance. Your algorithm cannot use more than $O(k)$ additional storage.*

Solution 10.5: We use a min-heap of size k. When the first k elements have been read in, the root holds the k-th largest element. Each successive element s is compared with the minimum element m in the heap. If s is less than or equal to m, do nothing. Otherwise, we remove m, and add s. The new root (which may or may not be s) is the k-th largest element. For the first k iterations, we simply add elements to the min-heap; the root holds the smallest value.

The time complexity per element processed is dominated by the time to delete the root, i.e., $O(\log k)$. The worst-case input is one in which elements appear in increasing order; the best case input is one in which elements appear in decreasing order.

```
template <typename T>
void find_k_th_largest_stream(istringstream &sin, const int &k) {
  priority_queue<T, vector<T>, greater<T>> min_heap;
  // The first k elements, output the minimum element
  T x;
  for (int i = 0; i < k && sin >> x; ++i) {
    min_heap.emplace(x);
    cout << min_heap.top() << endl;
  }

  // After the first k elements, output the k-th largest one
  while (sin >> x) {
    if (min_heap.top() < x) {
      min_heap.pop();
      min_heap.emplace(x);
    }
    cout << min_heap.top() << endl;
  }
}
```

Problem 10.6, *pg. 82*: *The input consists of a very long sequence of numbers. Each number is at most k positions away from its correctly sorted position. Design an algorithm that outputs the numbers in the correct order and uses $O(k)$ storage, independent of the number of elements processed.*

Solution 10.6: The easiest way of looking at this problem is that we need to store the numbers in memory till all the numbers smaller than this number have arrived. Once those numbers have arrived and have been written to the output file, we can go ahead and write this number. Since we do not know precisely what order the numbers appear in, it is not possible to say when all the numbers smaller than a given number have arrived and have been written to the output. However since

we are told that no number is off by more than k positions from its correctly sorted position, if more than k numbers greater than a given number have arrived and all the numbers smaller than the given number that arrived have been written, we can be sure that there are no more other smaller numbers that are going to arrive. Hence it is safe to write the given numbers.

This essentially gives us the strategy to always keep $k+1$ numbers in a min-heap. As soon as we read a new number, we extract the min from the heap and write the output and then insert the new number.

```
template <typename T>
void approximate_sort(istringstream &sin, const int &k) {
  priority_queue<T, vector<T>, greater<T>> min_heap;
  // Firstly push k elements into min_heap
  T x;
  for (int i = 0; i < k && sin >> x; ++i) {
    min_heap.push(x);
  }

  // Extract the minimum one for every incoming element
  while (sin >> x) {
    min_heap.push(x);
    cout << min_heap.top() << endl;
    min_heap.pop();
  }

  // Extract the remaining elements in min_heap
  while (min_heap.size()) {
    cout << min_heap.top() << endl;
    min_heap.pop();
  }
}
```

Problem 10.7, *pg. 82*: *Design an $O(n)$ time algorithm to compute the k elements closest to the median of an array A.*

Solution 10.7: There exists two standard algorithms for computing the median in $O(n)$ time—one uses randomized partitioning of the array; the other uses divide and conquer, specifically, it computes the median of the medians of $\lceil n/5 \rceil$ subarrays.

Assuming that we have computed the median μ in $O(n)$ time, we can compute the k elements closest to μ by maintaining a max-heap H of elements of the array. The value associated with the i-th element $A[i]$ is its distance to the median, i.e., $|\mu - A[i]|$. We start by adding the first k elements of the array to H. Now we process the remaining elements. For $j = k$ to $n-1$, if $|\mu - A[j]|$ is larger than the maximum value stored in the heap, we ignore it; otherwise, we remove the maximum element of H, and insert $A[j]$ in its place. When all elements are processed, the heap contains the k elements closest to the median.

Another approach, which does not require the $O(k)$ additional storage entailed by the max-heap, and runs in $O(n)$ time instead of $O(n \log k)$ time is to first compute the median μ, and then use a selection algorithm.

A selection algorithm takes as inputs a set A of n numbers, and an integer $i \in [1, n]$ and returns the i-th smallest element of A. There exists a practical selection algorithm, similar to quicksort, which runs in $O(n)$ expected time.

If we take $|A[j] - \mu|$ as the value of $A[j]$, and run the selection algorithm with $i = k$, we will get an element p. Let S be the elements strictly less than p. Suppose $|S| = k-1$. Then $\{p\} \cup S$ is the result. If $|S| < k - 1$, at least $k - |S|$ duplicates of p are present, in which case the union of any $k - |S|$ elements whose value is p with S is the result.

Note that both approaches start by computing the median, which changes the original array.

```
// Promote to double to prevent precision error
template <typename T>
double find_median(vector<T> &A) {
  int half = A.size() >> 1;
  nth_element(A.begin(), A.begin() + half, A.end());
  if (A.size() & 1) {  // A has odd number elements
    return A[half];
  } else {  // A has even number elements
    T x = A[half];
    nth_element(A.begin(), A.begin() + half - 1, A.end());
    return 0.5 * (x + A[half - 1]);
  }
}

template <typename T>
class Comp {
  private:
    double m_;

  public:
    Comp(const double &m) : m_(m) {};

    const bool operator()(const T &a, const T &b) const {
      return fabs(a - m_) < fabs(b - m_);
    }
};

template <typename T>
vector<T> find_k_closest_to_median(vector<T> A, const int &k) {
  // Find the element i where |A[i] - median| is k-th smallest
  nth_element(A.begin(), A.begin() + k - 1, A.end(), Comp<T>{find_median(A)});
  return {A.cbegin(), A.cbegin() + k};
}
```

Problem 10.8, *pg. 82*: *Design an algorithm for computing the running median of a sequence. The time complexity should be $O(\log n)$ per element read in, where n is the number of values read in up to that element.*

Solution 10.8: We use two heaps, L, a max-heap, and H, a min-heap. The invariant here is that for every incoming element from the stream, we want to let L store the smaller half of the stream data so far, and let H store the bigger half. By keeping this

invariant, we can output the median easily according to the number of elements we have seen so far. Following is the implementation in C++:

```
template <typename T>
void online_median(istringstream &sin) {
  // Min-heap stores the bigger part of the stream
  priority_queue<T, vector<T>, greater<T>> H;
  // Max-heap stores the smaller part of the stream
  priority_queue<T, vector<T>, less<T>> L;

  T x;
  while (sin >> x) {
    if (L.empty() == false && x > L.top()) {
      H.emplace(x);
    } else {
      L.emplace(x);
    }
    if (H.size() > L.size() + 1) {
      L.emplace(H.top());
      H.pop();
    } else if (L.size() > H.size() + 1) {
      H.emplace(L.top());
      L.pop();
    }

    if (H.size() == L.size()) {
      cout << 0.5 * (H.top() + L.top()) << endl;
    } else {
      cout << (H.size() > L.size() ? H.top() : L.top()) << endl;
    }
  }
}
```

Problem 10.9, *pg. 82*: *Design an algorithm for efficiently computing the k smallest real numbers of the form $a + b\sqrt{2}$ for nonnegative integers a and b.*

Solution 10.9: We can solve this problem using a min-heap H and a set S as follows. We initialize H to contain $0 + 0\sqrt{2} = 0$, and initialize S to the empty set. (A simple list will suffice to represent S). We now iteratively do the following, stopping when S has k elements. When we perform an extract-min from H to obtain a number $a + b\sqrt{2}$, we add it to H, and compute $c_1 = (a+1) + b\sqrt{2}$ and $c_2 = a + (b+1)\sqrt{2}$ which we add to H.

Suppose for the sake of contradiction that S is not the desired set. Since $|S| = k$, there has to be at least one number in the desired set that is not in S. Let the smallest such number be $m = p + q\sqrt{2}$. Note that p and q cannot both be 0. Similarly, there must be a number l that is in S and is greater than all numbers in S_2^k. If $p > 0$, consider the number $n = (p-1) + q\sqrt{2}$. It is less than m, and greater than 0, so it must be in S, since S contains all numbers in the desired set that are smaller than m. But then when we processed n to put it in S, we would have added n to H. This contradicts our adding l to S—the heap would always return n before l.

It is possible for a number to be inserted twice into the heap. For example, both $1 + 2\sqrt{2}$ and $2 + \sqrt{2}$ produce $2 + 2\sqrt{2}$. No number can be inserted more than twice: the irrationality of $\sqrt{2}$ implies that $a + b\sqrt{2} = c + d\sqrt{2}$ iff $a = b$ and $c = d$. We can check for duplicates when we perform extract-min.

```
class Num {
  public:
    int a_, b_;
    double val_;

    Num(const int &a, const int &b) : a_(a), b_(b), val_(a + b * sqrt(2)) {}

    const bool operator<(const Num &n) const {
      return val_ > n.val_;
    }

    // Equal function for hash
    const bool operator==(const Num &n) const {
      return a_ == n.a_ && b_ == n.b_;
    }
};

// Hash function for Num
class HashNum {
  public:
    const size_t operator()(const Num &n) const {
      return hash<int>()(n.a_) ^ hash<int>()(n.b_);
    }
};

vector<Num> generate_first_k(const int &k) {
  priority_queue<Num, vector<Num>> min_heap;
  vector<Num> smallest;
  unordered_set<Num, HashNum> hash;

  // Initial for 0 + 0 * sqrt(2)
  min_heap.emplace(0, 0);
  hash.emplace(0, 0);

  while (smallest.size() < k) {
    Num s(min_heap.top());
    smallest.emplace_back(s);
    hash.erase(s);
    min_heap.pop();

    // Add the next two numbers derived from s
    Num c1(s.a_ + 1, s.b_), c2(s.a_, s.b_ + 1);
    if (hash.emplace(c1).second) {
      min_heap.emplace(c1);
    }
    if (hash.emplace(c2).second) {
      min_heap.emplace(c2);
    }
  }
```

```
  return smallest;
}
```

Problem 10.10, *pg. 83*: *Design an $O(k)$ time algorithm for determining whether the k-th largest element in a max-heap is smaller than, equal to, or larger than a given x. The max-heap is represented using an array. Your algorithm's time complexity should be independent of the number of elements in the max-heap, and may use $O(k)$ additional storage. It cannot make any changes to the max-heap, and should handle the possibility of duplicate entries.*

Solution 10.10: We count the number of elements that are greater than or equal to x. The key to achieving an $O(k)$ time complexity is visiting max-heap nodes in best-first order, and stopping the computation as soon as we have found more than k nodes greater than x.

We use two integer variables, `equal` and `larger`, which are initialized to 0 and passed by reference to recursive calls of the check function. If the element r at the root is smaller than x, we know there are no elements in the max-heap larger than x, so we return right away. Otherwise, if $r = x$ we increment the `equal` count by 1; if $r > x$ we increment the `larger` count by 1. We then recurse on the left child and the right child. At any stage, if we determine there are more than k keys greater than x or more than k keys equal to x we return. Within each call we do constant work, and for each recursive call, either we increment `equal` or `larger`, or we came from a call that performed such an increment, implying the number of recursive calls is $O(k)$.

```
template <typename T>
void compare_k_th_largest_heap_helper(const vector<T> &max_heap, const int &k,
                                      const T &x, const int &idx, int &larger,
                                      int &equal) {
  if (idx < max_heap.size()) {
    if (max_heap[idx] < x) {
      return;
    } else if (max_heap[idx] == x) {
      ++equal;
    } else {  // max_heap[idx] > x
      ++larger;
    }

    if (equal < k && larger < k) {
      compare_k_th_largest_heap_helper(max_heap, k, x, (idx << 1) + 1, larger,
                                       equal);
      compare_k_th_largest_heap_helper(max_heap, k, x, (idx << 1) + 2, larger,
                                       equal);
    }
  }
}

// -1 means smaller, 0 means equal, and 1 means larger
template <typename T>
int compare_k_th_largest_heap(const vector<T> &max_heap, const int &k,
                              const T &x) {
  int larger = 0, equal = 0;
```

```
  compare_k_th_largest_heap_helper(max_heap, k, x, 0, larger, equal);
  return larger >= k ? 1 : (larger + equal >= k ? 0 : -1);
}
```

Problem 11.1, *pg. 86*: *Write a method that takes a sorted array A and a key k and returns the index of the* first *occurrence of k in A. Return −1 if k does not appear in A. For example, when applied to the array in Figure 11.1 on Page 86 your algorithm should return* 3 *if* $k = 108$*; if* $k = 285$*, your algorithm should return* 6*.*

Solution 11.1: The key idea is to search for k. However, even if we find k, after recording this we continue the search on the left subarray. The complexity bound is still $O(\log n)$—this is because each iteration reduces the size of the subarray being searched by half. In C++ code:

```
template <typename T>
int search_first(const vector<T> &A, const T &k) {
  int l = 0, r = A.size() - 1, res = -1;
  while (l <= r) {
    int m = l + ((r - l) >> 1);
    if (A[m] > k) {
      r = m - 1;
    } else if (A[m] == k) {
      // Record the solution and keep searching the left part
      res = m, r = m - 1;
    } else {  // A[m] < k
      l = m + 1;
    }
  }
  return res;
}
```

ϵ-Variant 11.1.1: Let A be an unsorted array of n integers, with $A[0] \geq A[1]$ and $A[n-2] \leq A[n-1]$. Call an index i a *local minimum* if $A[i]$ is less than or equal to its neighbors. How would you efficiently find a local minimum, if one exists?

ϵ-Variant 11.1.2: A sequence is said to be ascending if each element is greater than or equal to its predecessor; a descending sequence is one in which each element is less than or equal to its predecessor. A sequence is strictly ascending if each element is greater than its predecessor. Suppose it is known that an array A consists of an ascending sequence followed by a descending sequence. Design an algorithm for finding the maximum element in A. Solve the same problem when A consists of a strictly ascending sequence, followed by a descending sequence.

Problem 11.2, *pg. 86*: *Design an efficient algorithm that takes a sorted array A and a key k, and finds the index of the* first *occurrence an element larger than k; return −1 if every element is less than or equal to k. For example, when applied to the array in Figure 11.1 on Page 86 your algorithm should return −1 if* $k = 500$*; if* $k = 101$*, your algorithm should return* 3*.*

Solution 11.2: The naïve approach is to look for k via a binary search and then, if k is found, walk the array forward until either the first element larger than k is encountered or the end of the array is reached. If k is not found, binary search will end up pointing to either the first value greater than k in the array, in which case no further action is required or the last value smaller than k in which case the next element, if it exists, is the value that we are looking for. The worst-case run time of this algorithm is $\Theta(n)$—an array all of whose values equal k, except for the last one (which is greater than k), is the worst-case.

A better approach is to use binary search to eliminate half the candidates at each iteration: if we encounter an element larger than k, we record that element, and continue the search in the candidates on the left; otherwise, we continue searching in the candidates on the right. This algorithm has an $O(\log n)$ time complexity.

```
template <typename T>
int search_first_larger_k(const vector<T> &A, const T &k) {
  int l = 0, r = A.size() - 1, res = -1;
  while (l <= r) {
    int m = l + ((r - l) >> 1);
    if (A[m] > k) {
      // Record the solution and keep searching the left part
      res = m, r = m - 1;
    } else {  // A[m] <= k
      l = m + 1;
    }
  }
  return res;
}
```

Remark: As with Problem 11.1 on Page 86, the same problem can be posed for BSTs, and again, the solution is analogous to the one given above.

Problem 11.3, *pg. 86*: *Design an efficient algorithm that takes a sorted array A of distinct integers, and returns an index i such that $A[i] = i$ or indicate that no such index exists by returning -1. For example, when the input is the array shown in in Figure 11.1 on Page 86, your algorithm should return 2.*

Solution 11.3: Since the array contains distinct integers and is sorted, for any $i > 0$, $A[i] \geq A[i-1]+1$. Therefore $B[i] = A[i]-i$ is sorted. It follows that we can do a binary search for 0 in B to find an index such that $A[i] = i$. (We do not need to actually create B, we can simply use $A[i] - i$ wherever $B[i]$ is referenced.)

```
int search_index_value_equal(const vector<int> &A) {
  int l = 0, r = A.size() - 1;
  while (l <= r) {
    int m = l + ((r - l) >> 1);
    int val = A[m] - m;
    if (val == 0) {
      return m;
    } else if (val > 0) {
      r = m - 1;
    } else {  // val < 0
```

```
      l = m + 1;
    }
  }
  return -1;
}
```

Variant 11.3.1: Solve the same problem when A is sorted but may contain duplicates.

Problem 11.4, *pg. 87*: *Design an algorithm that takes an abs-sorted array A and a number k, and returns a pair of indices of elements in A that sum up to k. For example, if the input to your algorithm is the array in Figure 11.2 on Page 87 and $k = 167$, your algorithm should output $(3, 7)$. Output $(-1, -1)$ if there is no such pair.*

Solution 11.4: First consider the case where the array is sorted in the conventional sense. In this case we can start with the pair consisting of the first element and the last element: $(A[0], A[n-1])$. Let $s = A[0] + A[n-1]$. If $s = k$, we are done. If $s < k$, we increase the sum by moving to pair $(A[1], A[n-1])$. We need never consider $A[0]$; since the array is sorted, for all i, $A[0] + A[i] \leq A[0] + A[n-1] = k < s$. If $s > k$, we can decrease the sum by considering the pair $(A[0], A[n-2])$; by analogous reasoning, we need never consider $A[n-1]$ again. We iteratively continue this process till we have found a pair that sums up to k or the indices meet, in which case the search ends. This solution works in $O(n)$ time and $O(1)$ space in addition to the space needed to store A.

This approach will not work when the array entries are sorted by absolute value. In this instance, we need to consider three cases:

(1.) Both the numbers in the pair are negative.
(2.) Both the numbers in the pair are positive.
(3.) One is negative and the other is positive.

For Cases (1.) and (2.), we can run the above algorithm separately by just limiting ourselves to either positive or negative numbers. For Case (3.), we can use the same approach where we have one index for positive numbers, one index for negative numbers, and they both start from the highest possible index and then go down.

```
template <typename T, typename Comp>
pair<int, int> find_pair_using_comp(const vector<T> &A, const T &k,
                                    Comp comp) {
  pair<int, int> ret(0, A.size() - 1);
  while (ret.first < ret.second && comp(A[ret.first], 0)) {
    ++ret.first;
  }
  while (ret.first < ret.second && comp(A[ret.second], 0)) {
    --ret.second;
  }

  while (ret.first < ret.second) {
    if (A[ret.first] + A[ret.second] == k) {
      return ret;
    } else if (comp(A[ret.first] + A[ret.second], k)) {
```

```
      do {
        ++ret.first;
      } while (ret.first < ret.second && comp(A[ret.first], 0));
    } else {
      do {
        --ret.second;
      } while (ret.first < ret.second && comp(A[ret.second], 0));
    }
  }
  return {-1, -1};  // no answer
}

template <typename T>
pair<int, int> find_pos_neg_pair(const vector<T> &A, const T &k) {
  // ret.first for positive, and ret.second for negative
  pair<int, int> ret(A.size() - 1, A.size() - 1);
  // Find the last positive or zero
  while (ret.first >= 0 && A[ret.first] < 0) {
    --ret.first;
  }

  // Find the last negative
  while (ret.second >= 0 && A[ret.second] >= 0) {
    --ret.second;
  }

  while (ret.first >= 0 && ret.second >= 0) {
    if (A[ret.first] + A[ret.second] == k) {
      return ret;
    } else if (A[ret.first] + A[ret.second] > k) {
      do {
        --ret.first;
      } while (ret.first >= 0 && A[ret.first] < 0);
    } else {  // A[ret.first] + A[ret.second] < k
      do {
        --ret.second;
      } while (ret.second >= 0 && A[ret.second] >= 0);
    }
  }
  return {-1, -1};  // no answer
}

template <typename T>
pair<int, int> find_pair_sum_k(const vector<T> &A, const T &k) {
  pair<int, int> ret = find_pos_neg_pair(A, k);
  if (ret.first == -1 && ret.second == -1) {
    return k >= 0 ? find_pair_using_comp(A, k, less<T>()) :
           find_pair_using_comp(A, k, greater_equal<T>());
  }
  return ret;
}
```

A simpler solution is based on a hash table (Chapter 12) to store all the numbers and then for each number x in the array, look up $k - x$ in the hash table. If the

hash function does a good job of spreading the keys, the time complexity for this approach is $O(n)$. However, it requires $O(n)$ additional storage. If the array is sorted on elements (and not absolute values), for each $A[i]$ we can use binary search to find $k-A[i]$. This approach uses $O(1)$ additional space and has time complexity $O(n \log n)$. However, it is strictly inferior to the two pointer technique described at the beginning of the solution.

Variant 11.4.1: Design an algorithm that takes as input an array of integers A, and an integer k, and returns a pair of indices i and j such that $A[j] - A[i] = k$, if such a pair exists.

Problem 11.5, *pg. 87*: *Design an $O(\log n)$ algorithm for finding the position of the smallest element in a cyclically sorted array. Assume all elements are distinct. For example, for the array in Figure 11.3 on Page 87, your algorithm should return 4.*

Solution 11.5: We make use of the decrease and conquer principle. Specifically, we maintain an interval of candidate indices, and iteratively eliminate a constant fraction of the indices in this interval. Let $I = [l_I, r_I]$ be the set of indices being considered, and m_I be the midpoint of I, i.e., $l_I + \lfloor \frac{r_I - l_I}{2} \rfloor$. If $A[m_I] > A[r_I]$ then $[l_I, m_I]$ cannot contain the index of the minimum element. Therefore we can restrict the search to $[m_I + 1, r_I]$. If $A[m_I] < A[r_I]$ we restrict our attention to $[l_I, m_I]$. We start with $I = [0, n-1]$, and end when the interval has one element.

```
template <typename T>
int search_smallest(const vector<T> &A) {
  int l = 0, r = A.size() - 1;
  while (l < r) {
    int m = l + ((r - l) >> 1);
    if (A[m] > A[r]) {
      l = m + 1;
    } else {  // A[m] <= A[r]
      r = m;
    }
  }
  return l;
}
```

Note that this problem cannot be solved in less than linear time when elements may be repeated. For example, if A consists of $n-1$ 1s and a single 0, that 0 cannot be detected in the worst case without inspecting every element. Following is the C++ code for the scenario when elements may be repeated:

```
template <typename T>
int search_smallest_helper(const vector<T> &A, const int &l, const int &r) {
  if (l == r) {
    return l;
  }

  int m = l + ((r - l) >> 1);
  if (A[m] > A[r]) {
```

```
    return search_smallest_helper(A, m + 1, r);
  } else if (A[m] < A[r]) {
    return search_smallest_helper(A, l, m);
  } else {  // A[m] == A[r]
    // Smallest element must exist in either left or right side
    int l_res = search_smallest_helper(A, l, m);
    int r_res = search_smallest_helper(A, m + 1, r);
    return A[r_res] < A[l_res] ? r_res : l_res;
  }
}

template <typename T>
int search_smallest(const vector<T> &A) {
  return search_smallest_helper(A, 0, A.size() - 1);
}
```

Variant 11.5.1: Design an $O(\log n)$ algorithm for finding the position of an element k in a cyclically sorted array.

Problem 11.6, *pg. 87*: *Let A be a sorted array. The length of A is not known in advance; accessing $A[i]$ for i beyond the end of the array throws an exception. Design an algorithm that takes A and a key k and returns an index i such that $A[i] = k$; return -1 if k does not appear in A.*

Solution 11.6: The key idea here is to simultaneously do a binary search for the end of the array as well as the key. We examine $A[2^p - 1]$ in the p-th step till we hit an exception or an entry greater than or equal to k. Then we do a conventional binary search for k in the range $[2^{p-1} + 1, 2^p - 2]$. The run time of the search algorithm is $O(\log n)$, where n is the length of A. In code:

```
template <typename T>
int binary_search_unknown_len(const vector<T> &A, const T &k) {
  // Find the possible range where k exists
  int p = 0;
  while (true) {
    try {
      T val = A.at((1 << p) - 1);
      if (val == k) {
        return (1 << p) - 1;
      } else if (val > k) {
        break;
      }
    }
    catch (exception& e) {
      break;
    }
    ++p;
  }

  // Binary search between indices 2^(p - 1) + 1 and 2^p - 2
  int l = (1 << (p - 1)) + 1, r = (1 << p) - 2;
  while (l <= r) {
```

```
    int m = l + ((r - l) >> 1);
    try {
      T val = A.at(m);
      if (val == k) {
        return m;
      } else if (val > k) {
        r = m - 1;
      } else {  // A[m] < k
        l = m + 1;
      }
    }
    catch (exception& e) {
      r = m - 1;  // search the left part if out of boundary
    }
  }
  return -1;  // nothing matched k
}
```

Problem 11.7, *pg. 88*: *Let A be an array of n nonnegative real numbers and S' be a nonnegative real number less than $\sum_{i=0}^{n-1} A[i]$. Design an efficient algorithm for computing σ such that $\sum_{i=0}^{n-1} \min(A[i], \sigma) = S'$, if such a σ exists.*

Solution 11.7: Define $F(\sigma) = \sum_{i=0}^{n-1} \min(A[i], \sigma)$. We are looking for a value of σ such that $F(\sigma) = S'$. Clearly, F is a continuous function of σ. Since $0 \leq S' \leq \sum_{i=0}^{n-1} A[i]$, by the intermediate value theorem of calculus, there must exist a value of σ in $[0, \max_{i=0}^{n-1} A[i]]$ such that $F(\sigma) = S'$. Furthermore, since F monotonically increases with σ, we can perform binary search on the interval $[0, \max_{i=0}^{n-1} A[i]]$ to find the correct value of σ.

Assume that A is already sorted, i.e., for all i, $A[i] \leq A[i+1]$. Compute the prefix sum $z_k = \sum_{i=0}^{k-1} A[i]$. Now, suppose $A[k-1] \leq \sigma \leq A[k]$. Consequently, $F(\sigma) = (n-k)\sigma + z_k$.

Using the above expression, we can search for the value of k such that $F(A[k]) \leq S' \leq F(A[k+1])$ by performing binary search for k. (Since we sort A, the run time of our solution is already $O(n \log n)$, implying we could do a linear search without changing the time complexity.) Once we have found the right value of k, we can compute the value of σ by simply solving the equation for $F(\sigma)$ above.

The most expensive operation for this entire solution is sorting A, hence the run time is $O(n \log n)$. If we are given A sorted in advance and its prefix sums, then for each value of S', the search would have time complexity $O(\log n)$.

```
double completion_search(vector<double> &A, const double &budget) {
  sort(A.begin(), A.end());
  // Calculate the prefix sum for A
  vector<double> prefix_sum;
  partial_sum(A.cbegin(), A.cend(), back_inserter(prefix_sum));
  // costs[i] represents the total payroll if the cap is A[i]
  vector<double> costs;
  for (int i = 0; i < prefix_sum.size(); ++i) {
    costs.emplace_back(prefix_sum[i] + (A.size() - i - 1) * A[i]);
  }

```

```
  auto lower = lower_bound(costs.cbegin(), costs.cend(), budget);
  if (lower == costs.cend()) {
    return -1.0;  // no solution since budget is too large
  }

  if (lower == costs.cbegin()) {
    return budget / A.size();
  }
  int idx = lower - costs.cbegin() - 1;
  return A[idx] + (budget - costs[idx]) / (A.size() - idx - 1);
}
```

ϵ-Variant 11.7.1: Solve the same problem using only $O(1)$ space.

Problem 11.8, *pg. 88*: *You are given two sorted arrays A and B of lengths m and n, respectively, and a positive integer $k \in [1, m+n]$. Design an algorithm that runs in $O(\log k)$ time for computing the k-th smallest element in array formed by merging A and B. Array elements may be duplicated within and between A and B.*

Solution 11.8: Suppose the first k elements of the union of A and B consist of the first x elements of A and the first $k-x$ elements of B. We'll use binary search to determine x.

Specifically, we will maintain an interval $[l, u]$ that contains x, and use binary search to iteratively half the size of the interval. At each iteration we try $x = \lfloor \frac{l+u}{2} \rfloor$. If $l = u$, we simply return the larger of $A[x-1]$ and $B[k-x-1]$. (If $A[x-1] = B[k-1-x]$, we arbitrarily return either.) If $l \neq u$ but $A[x-1] = B[k-1-x]$, we return $A[x-1]$, since the first x elements of A and the first $k-x$ elements of B when sorted end in $A[x-1]$ or $B[k-1-x]$, which are equal.

Otherwise, if $A[x] < B[k-x-1]$, then $A[x]$ must be in the first k elements of the union, so we can update l to $x+1$. Conversely, if $B[k-x] < A[x-1]$ then $A[x-1]$ cannot be in the first k elements, so we can update u to $x-1$.

The initial values for l and u need to be chosen carefully. Naïvely setting $l = 0, u = k$ does not work, since this choice may lead to x lying outside the range of valid indices for B, i.e., outside $[0, n-1]$. Setting $l = \max(0, k-n)$ and $u = \min(m, k)$ resolves this problem.

```
template <typename T>
T find_kth_in_two_sorted_arrays(const vector<T> &A, const vector<T> &B,
                                const int &k) {
  // Lower bound of elements we will choose in A
  int l = max(0, static_cast<int>(k - B.size()));
  // Upper bound of elements we will choose in A
  int u = min(static_cast<int>(A.size()), k);

  while (l < u) {
    int x = l + ((u - l) >> 1);
    T A_x_1 = (x <= 0 ? numeric_limits<T>::min() : A[x - 1]);
    T A_x = (x >= A.size() ? numeric_limits<T>::max() : A[x]);
    T B_k_x_1 = (k - x <= 0 ? numeric_limits<T>::min() : B[k - x - 1]);
```

```
    T B_k_x = (k - x >= B.size() ? numeric_limits<T>::max() : B[k - x]);

    if (A_x < B_k_x_1) {
      l = x + 1;
    } else if (A_x_1 > B_k_x) {
      u = x - 1;
    } else {
      // B[k - x - 1] <= A[x] && A[x - 1] < B[k - x]
      return max(A_x_1, B_k_x_1);
    }
  }

  T A_l_1 = l <= 0 ? numeric_limits<T>::min() : A[l - 1];
  T B_k_l_1 = k - l - 1 < 0 ? numeric_limits<T>::min() : B[k - l - 1];
  return max(A_l_1, B_k_l_1);
}
```

Problem 11.9, *pg. 88*: *Implement a function which takes as input a floating point variable x and returns $\sqrt{x}$.*

Solution 11.9: One of the fastest ways to invert a fast-growing monotone function (such as the square function) is to do a binary search. Given x, we start with a lower bound l and an upper bound u on $\sqrt{x}$. We iteratively check if the square of midpoint m of $[l, u]$ is smaller than, greater than, or equal to x. In the first case, we update the lower bound to m; in the second case, we update the upper bound to m; in the third case, we return m.

When checking for equality, we use a notion of tolerance, eps, since floating point arithmetic is not exact. This tolerance is user-specified.

Trivial choices for the initial lower bound and upper bound are 0 and the largest floating point number that is representable. If $x \geq 1.0$, we can tighten the lower and upper bounds to 1.0 and x, since $x \geq 1.0 \Rightarrow x^2 \geq x$. If $x < 1.0 \Rightarrow x^2 < x$, the previous choice of l and u is incorrect; instead, we can use x and 1.0. The time complexity is $O(\log \frac{x}{\text{eps}})$ since the number of iterations is affected by the choice of eps. Care has to be taken to ensure the compare function is resilient to finite precision effects.

```
// 0 means equal, -1 means smaller, 1 means larger
int compare(const double &a, const double &b) {
  // Use normalization for precision problem
  double diff = (a - b) / b;
  return diff < -numeric_limits<double>::epsilon() ?
         -1 : diff > numeric_limits<double>::epsilon();
}

double square_root(const double &x) {
  // Decide the search range according to x
  double l, r;
  if (compare(x, 1.0) < 0) {  // x < 1.0
    l = x, r = 1.0;
  } else {  // x >= 1.0
    l = 1.0, r = x;
  }
```

```

  // Keep searching if l < r
  while (compare(l, r) == -1) {
    double m = l + 0.5 * (r - l);
    double square_m = m * m;
    if (compare(square_m, x) == 0) {
      return m;
    } else if (compare(square_m, x) == 1) {
      r = m;
    } else {
      l = m;
    }
  }
  return l;
}
```

Variant 11.9.1: Given two positive floating point numbers x and y, how would you compute $\frac{x}{y}$ to within a specified tolerance ϵ if the division operator cannot be used? You cannot use any library functions, such as *log* and *exp*; addition and multiplication are acceptable.

Problem 11.10, *pg. 88*: *Let A be an $n \times n$ 2D array where rows and columns are sorted in increasing sorted order. Design an efficient algorithm that decides whether a number x appears in A. How many entries of A does your algorithm inspect in the worst-case? Can you prove a tight lower bound that any such algorithm has to consider in the worst-case?*

Solution 11.10: One approach is to start by comparing x to $A[0][n-1]$. If $x = A[0][n-1]$, stop. Otherwise:

- $x > A[0][n-1]$, in which case x is greater than all elements in Row 0.
- $x < A[0][n-1]$, in which case x is less than all elements in Column $n-1$.

In either case, we have a 2D array with n fewer elements to search. In each iteration, we remove a row or a column, which means we inspect at most $2n-1$ elements.

```
template <typename T>
bool matrix_search(const vector<vector<T>> &A, const T &x) {
  int r = 0, c = A[0].size() - 1;
  while (r < A.size() && c >= 0) {
    if (A[r][c] == x) {
      return true;
    } else if (A[r][c] < x) {
      ++r;
    } else {  // A[r][c] > x
      --c;
    }
  }
  return false;
}
```

For a tight lower bound, let x be any input. Define A to be

$$\begin{bmatrix} & & & & x-1 \\ & & & & x+1 \\ & & & \cdots & \\ & & x-1 & & \\ & x-1 & x+1 & & \\ x-1 & x+1 & & & \\ x+1 & & & & \end{bmatrix}$$

where entries not shown are chosen so that the matrix is sorted by rows and by columns. We claim that any algorithm that solves the 2D array search problem will have to compare x with each of the $2n-1$ elements shown (i.e., the diagonal elements and the elements immediately below them). Call these elements the Δ elements.

Proof:

Comparing x with other elements does not eliminate any of the Δ elements. Suppose an algorithm did not compare x with one of the Δ elements. Then we could make that element x (instead of $x-1$ or $x+1$) and the algorithm would behave exactly as before and hence return the wrong result. Therefore at least $2n-1$ compares are necessary which means that the algorithm we designed is optimum.

Problem 11.11, *pg. 89*: *How would you organize a tournament with 128 players to minimize the number of matches needed to find the best player? How many matches do you need to find the best and the second best player?*

Solution 11.11: First, we consider the problem of finding the best player. Each game eliminates one player and there are 128 players; so, 127 matches are necessary and also sufficient.

To find the second best, we note that the only candidates are the players who are beaten by the player who is eventually determined to be the best—everyone else lost to someone who is not the best.

To find the best player, the order in which we organize the matches is inconsequential—we just pick pairs from the set of candidates and whoever loses is removed from the pool of candidates. However if we proceed in an arbitrary order, we might start with the best player, who defeats 127 other players and then the players who lost need to play 126 matches amongst themselves to find the second best.

To find the second best player, we can do much better by organizing the matches as a complete binary tree. Specifically, we pair off all the players arbitrarily to form 64 matches. After these matches, we are left with 64 candidates; we pair them off again arbitrarily and they play 32 matches. Proceeding in this fashion, we organize the 127 matches needed to find the best player and the winner will have played 7 matches. Therefore we can find the second best player by organizing 6 matches between the 7

players who lost to the best player, for a total of $127 + (7 - 1) = 133$ matches.

Problem 11.12, *pg. 89*: *Find the min and max elements from an array of n elements using no more than $\lceil 3n/2 \rceil - 2$ comparisons.*

Solution 11.12: If $n = 1$, no comparisons are needed. Suppose $n > 1$. Find the min m and the max M of the first two numbers; this requires a single comparison. Now process the remaining elements two at a time. Let (x, y) be such a pair. If $\min(x, y) < m$, update m; if $\max(x, y) > M$, update M. This entails three comparisons for each pair. If n is odd, the last update entails two comparisons, namely comparing the last element with m and M.

```
// Return (min, max) pair of elements in A
template <typename T>
pair<T, T> find_min_max(const vector<T> &A) {
  if (A.size() <= 1) {
    return {A.front(), A.front()};
  }

  // Initialize the min and max pair
  pair<T, T> min_max_pair = minmax(A[0], A[1]);
  for (int i = 2; i + 1 < A.size(); i += 2) {
    pair<T, T> local_pair = minmax(A[i], A[i + 1]);
    min_max_pair = {min(min_max_pair.first, local_pair.first),
                    max(min_max_pair.second, local_pair.second)};
  }
  // Special case: if there is odd number of elements in the array, we still
  // need to compare the last element with the existing answer.
  if (A.size() & 1) {
    min_max_pair = {min(min_max_pair.first, A.back()),
                    max(min_max_pair.second, A.back())};
  }
  return min_max_pair;
}
```

Variant 11.12.1: What is the least number of comparisons required to find the min and the max in the worst case?

Problem 11.13, *pg. 89*: *Design an algorithm for computing the k-th largest element in an array A that runs in $O(n)$ expected time.*

Solution 11.13: The basic idea is to use decrease and conquer. We pick a random index r in the array A. Let $A[r] = x$. Reorder the elements in A in such a way that all elements that appear before index p are greater than x, and all elements that appear after p are less than or equal to x. Call the reordered array A'.

If $p = k - 1$, we are done—$A'[p]$ is the k-th largest element. Otherwise if $p > k - 1$, the element we are looking for is the k-th largest element of the subarray$A'[0 : p - 1]$. Finally, if $p < k - 1$, the element we seek is the $(k - (p + 1))$-th largest element of the subarray $A'[p + 1 : n - 1]$. Each of the two latter cases can be solved recursively.

Since we expect to split the array into roughly equal halves on average, intuitively, the expected time complexity $T(n)$ should satisfy $T(n) = O(n) + T(n/2)$. This solves to $T(n) = O(n)$. A more formal analysis requires the use of indicator random variables X_i, for $0 \le i \le k-1$, one per choice of t, and leads to the same conclusion.

```
// Partition A according pivot, return its index after partition
template <typename T>
int partition(vector<T> &A, const int &l, const int &r, const int &pivot) {
  T pivot_value = A[pivot];
  int larger_index = l;

  swap(A[pivot], A[r]);
  for (int i = l; i < r; ++i) {
    if (A[i] > pivot_value) {
      swap(A[i], A[larger_index++]);
    }
  }
  swap(A[r], A[larger_index]);
  return larger_index;
}

template <typename T>
T find_k_th_largest(vector<T> A, const int &k) {
  int l = 0, r = A.size() - 1;

  while (l <= r) {
    default_random_engine gen((random_device())());
    uniform_int_distribution<int> dis(l, r); // generate random int in [l, r]
    int p = partition(A, l, r, dis(gen));
    if (p == k - 1) {
      return A[p];
    } else if (p > k - 1) {
      r = p - 1;
    } else {  // p < k - 1
      l = p + 1;
    }
  }
}
```

ϵ-Variant 11.13.1: Design an algorithm for finding the k-th largest element of A in the presence of duplicates. The k-th largest element is defined to be $A[k-1]$ after A has been sorted in a stable manner, i.e., if $A[i] = A[j]$ and $i < j$ then $A[i]$ must appear before $A[j]$ after stable sorting.

Problem 11.14, *pg. 89*: *Design an algorithm for computing the k-th largest element in a sequence of elements. It should run in $O(n)$ expected time where n is the length of the sequence, which is not known in advance. The value k is known in advance. Your algorithm should print the k-th largest element after the sequence has ended. It should use $O(k)$ additional storage.*

Solution 11.14: The natural approach is to use a min-heap containing the k largest

elements seen thus far. As each new element e is read, it is compared with the value of the smallest element m in the min-heap: if $e \leq m$, we continue; otherwise we delete m and insert e. This approach has time complexity $O(n \log k)$, since inserts and deletes take $O(\log k)$ time, and when elements are presented in ascending order, each new element requires an insert and a delete.

A better approach is to keep the k largest elements in an array M of length $2k - 1$. We add elements to M, and each time M is full, we find the k largest elements using the selection algorithm in Solution 11.13 on Page 270. The smaller elements are discarded, and we continue. The selection algorithm takes $O(k)$ time and is run every k elements, implying an $O(n)$ time complexity.

```
template <typename T>
T find_k_th_largest_unknown_length(istringstream &sin, const int &k) {
  vector<T> M;
  T x;
  while (sin >> x) {
    M.emplace_back(x);
    if (M.size() == (k << 1) - 1) {
      // Keep the k largest elements and discard the small ones
      nth_element(M.begin(), M.begin() + k - 1, M.end(), greater<T>());
      M.resize(k);
    }
  }
  nth_element(M.begin(), M.begin() + k - 1, M.end(), greater<T>());
  return M[k - 1];  // return the k-th largest one
}
```

Problem 11.15, *pg. 90*: *Suppose you were given a file containing roughly one billion Internet Protocol (IP) addresses, each of which is a 32-bit unsigned integer. How would you programmatically find an IP address that is not in the file? Assume you have unlimited drive space but only two megabytes of RAM at your disposal.*

Solution 11.15: In the first step, we build an array of 2^{16} 32-bit unsigned integers that is initialized to 0 and for every IP address in the file, we take its 16 most significant bits to index into this array and increment the count of that number. Since the file contains fewer than 2^{32} numbers, there must be one entry in the array that is less than 2^{16}. This tells us that there is at least one IP address which has those upper bits and is not in the file. In the second pass, we can focus only on the addresses that match this criterion and use a bit array of size 2^{16} to identify one of the missing numbers.

```
int find_missing_element(ifstream &ifs) {
  vector<size_t> counter(1 << 16, 0);
  unsigned int x;
  while (ifs >> x) {
    ++counter[x >> 16];
  }

  for (int i = 0; i < counter.size(); ++i) {
    // Find one bucket contains less than (1 << 16) elements
    if (counter[i] < (1 << 16)) {
```

```
      bitset<(1 << 16)> bit_vec;
      ifs.clear();
      ifs.seekg(0, ios::beg);
      while (ifs >> x) {
        if (i == (x >> 16)) {
          bit_vec.set(((1 << 16) - 1) & x);  // gets the lower 16 bits of x
        }
      }
      ifs.close();

      for (int j = 0; j < (1 << 16); ++j) {
        if (bit_vec.test(j) == 0) {
          return (i << 16) | j;
        }
      }
    }
  }
}
```

Problem 11.16, pg. 90: *Let A be an array of n integers in $\mathcal{Z}_n$, with exactly one element t appearing twice. This implies exactly one element $m \in \mathcal{Z}_n$ is missing from A. How would you compute t and m in $O(n)$ time and $O(1)$ space?*

Solution 11.16: Let $\text{Sum}(\mathcal{Z}_n)$ be the sum of the elements in $\mathcal{Z}_n$, and $\text{Sqr}(\mathcal{Z}_n)$ be the sum of the squares of the elements in $\mathcal{Z}_n$. The sum of the elements in A is exactly $\text{Sum}(\mathcal{Z}_n) + t - m$, and the sum of the squares of the elements in A is exactly $\text{Sqr}(\mathcal{Z}_n) + t^2 - m^2$. It is straightforward to compute $m - t$: initialize sum to 0, and add $(i - A[i])$ to sum for each index i. A similar computation yields $m^2 - t^2$ (i.e., square_sum in code). Factoring and canceling the expression $\frac{m^2-t^2}{m-t}$ yields $m + t$, to which we add $m - t$ to obtain $2m$ and subtract $m - t$ to obtain $2t$. Details are given below:

```
// Return pair<int, int>(duplicate, missing)
pair<int, int> find_duplicate_missing(const vector<int> &A) {
  int sum = 0, square_sum = 0;
  for (int i = 0; i < A.size(); ++i) {
    sum += i - A[i], square_sum += i * i - A[i] * A[i];
  }
  return {(square_sum / sum - sum) >> 1, (square_sum / sum + sum) >> 1};
}
```

The problem with the approach above is that it can lead to overflow. A substantially better approach is to compute the XOR of all the elements in $\mathcal{Z}_n$ and A—this yields $m \oplus t$. Since $m \neq t$, there must be some bit in $m \oplus t$ that is set to 1, i.e., m and t differ in that bit. We then compute the XOR of all the elements in $\mathcal{Z}_n$ and in A in which that bit is 1. Let this XOR be h. By the logic described in the problem statement, h must be one of m or t. We can scan through A to determine if h is the duplicate or the missing element. This approach is simpler and, since it requires no arithmetic, it cannot result in an overflow. A disadvantage is that it requires three passes through A.

```
// Return pair<int, int>(duplicate, missing)
pair<int, int> find_duplicate_missing(const vector<int> &A) {
  int miss_XOR_dup = 0;
  for (int i = 0; i < A.size(); ++i) {
    miss_XOR_dup ^= i ^ A[i];
  }

  int differ_bit = miss_XOR_dup & (~(miss_XOR_dup - 1)), miss_or_dup = 0;
  for (int i = 0; i < A.size(); ++i) {
    if (i & differ_bit) {
      miss_or_dup ^= i;
    }
    if (A[i] & differ_bit) {
      miss_or_dup ^= A[i];
    }
  }

  for (const int &A_i : A) {
    if (A_i == miss_or_dup) {   // find duplicate
      return {miss_or_dup, miss_or_dup ^ miss_XOR_dup};
    }
  }
  // miss_or_dup is missing element
  return {miss_or_dup ^ miss_XOR_dup, miss_or_dup};
}
```

Problem 11.17, *pg. 90*: *Given an array A, in which each element of A appears three times except for one element e that appears once, find e in $O(1)$ space and $O(n)$ time.*

Solution 11.17: One way to view Solution 11.16 on the preceding page is that it counts modulo 2 for each bit-position the number of entries in which the bit in that position is 1. Specifically, the XOR of elements at indices $[0, i-1]$, determines exactly which bit-positions have been 1 an odd number of times in elements of A whose indices are in $[0, i-1]$.

The analogous approach for the current problem is to count modulo 3 for each bit-position the number of times the bit in that position has been 1. The effect of counting modulo 3 is to eliminate the elements that appear three times, and so the bit-positions which have a count of 1 are precisely those bit-positions in e which are set to 1.

Representing a number modulo 3 requires two bits. We use two integer-valued variables, `ones` and `twos`, to do the counting. The variable `ones` denotes whether a bit-position has been set once (modulo 3) so far; the variable `twos` denotes whether a bit-position has been set twice (modulo 3) so far. When a bit-position has a count of 2 (modulo 3) and another 1 is observed, we reset the `ones` and `twos` variables.

Suppose `ones` and `twos` have been set appropriately after reading in the first $i-1$ elements. After reading A_{i-1}, bit-position j has a count of 1 modulo 3 iff it had a count of 1 modulo 3 and the j-th bit in A_i is a zero or the count was 0 modulo 3 and the j-th bit in A_i is a one. This gives us the update equation for `ones`.

After reading A_{i-1}, bit-position j has a count of 2 modulo 3 iff it had a count of 2 modulo 3 and the j-th bit in A_{i-1} is a zero or the count was 1 modulo 3 and the j-th bit in i is a one. This gives us the update equation for twos.

The code below implements the update equations; for the reasons described above, the final result is ones.

```
int find_element_appears_once(const vector<int> &A) {
  int ones = 0, twos = 0;
  int next_ones, next_twos;
  for (const int &i : A) {
    next_ones = (~i & ones) | (i & ~ones & ~twos);
    next_twos = (~i & twos) | (i & ones);
    ones = next_ones, twos = next_twos;
  }
  return ones;
}
```

Problem 11.18, *pg. 91*: *Design an efficient algorithm that takes a close array A, and a key k and searches for any index j such that $A[j] = k$. Return -1 if no such index exists. For example, for the array in Figure 11.4 on Page 91, if $k = 2$, your algorithm should return an index in $\{4, 5, 7\}$.*

Solution 11.18: The close property allows us to skip indices: if $|A[i] - k| = l$, then for no index $i' \in (i - l, i + l)$ can $A[i'] = k$. We use this test to speedup the basic iterative search through an array in the code given below.

```
int close_search(const vector<int> &A, const int &k) {
  int idx = 0;
  while (idx < A.size() && A[idx] != k) {
    idx += abs(A[idx] - k);
  }
  return idx < A.size() ? idx : -1;  // -1 means no result
}
```

A worst-case input is one where all elements have value $k - 1$, in which case the algorithm is forced to inspect all elements. Hence its time complexity is $O(n)$, where n is the length of the close array. (Recall time complexity measures performance on worst-case inputs.) In the best case, our algorithm inspects $1/k$-th of the array, e.g., if all elements are 0.

Problem 11.19, *pg. 91*: *You are reading a sequence of words from a very long stream. You know* a priori *that more than half the words are repetitions of a single word w (the "majority element") but the positions where w occurs are unknown. Design an algorithm that makes a single pass over the stream and uses only a constant amount of memory to identify w.*

Solution 11.19: The following observation leads to an elegant solution. If you take any two distinct elements from the stream and discard them away, the majority element remains the majority of the remaining elements. (This hinges on the assumption that there exists a majority element to begin with). The reasoning is follows.

Proof:

Let's say the majority element occurred m times out of n elements in the stream such that $\frac{m}{n} > \frac{1}{2}$. The two distinct elements that are discarded can have at most one of the majority elements. Hence after discarding them, the ratio of the previously majority element to the total number of elements is either $\frac{m}{(n-2)}$ or $\frac{(m-1)}{(n-2)}$. It is simple to verify that if $\frac{m}{n} > \frac{1}{2}$, then $\frac{m}{(n-2)} > \frac{(m-1)}{(n-2)} > \frac{1}{2}$.

Now, as we read the stream from beginning to the end, as soon as we encounter more than one distinct element, we can discard one instance of each element and what we are left with in the end must be the majority element.

```
string majority_search(istringstream &sin) {
  string candidate, buf;
  int count = 0;
  while (sin >> buf) {
    if (count == 0) {
      candidate = buf;
      count = 1;
    } else if (candidate == buf) {
      ++count;
    } else {
      --count;
    }
  }
  return candidate;
}
```

The code above assumes a majority word exists in the sequence. If no word has a strict majority, it still returns a word from the stream, albeit without any meaningful guarantees on how common that word is. We could check with a second pass whether the returned word was a majority. Similar ideas can be used to identify words that appear more than n/k times in the sequence, as discussed in Problem 12.11 on Page 96.

Problem 12.1, *pg. 92*: *Design a hash function that is suitable for words in a dictionary.*

Solution 12.1: First, the hash function should examine all the characters in each word. (If this seem obvious, the string hash function in the original distribution of Java examined at most 16 characters, in an attempt to gain speed, but often resulted in very poor performance because of collisions.) It should give a large range of values, and should not let one character dominate (e.g., if we simply cast characters to integers and multiplied them, a single 0 would result in a hash code of 0). We would also like a rolling hash function, one in which if a character is deleted from the front of the string, and another added to the end, the new hash code can be computed in $O(1)$ time (see Solution 12.13 on Page 286). The following function has these properties:

```
int string_hash(const string &str, const int &modulus) {
  const int MULT = 997;
  int val = 0;
  for (const char &c : str) {
```

```
    val = (val * MULT + c) % modulus;
  }
  return val;
}
```

Problem 12.2, *pg. 93*: *Design a hash function for chess game states. Your function should take a state and the hash code for that state, and a move, and efficiently compute the hash code for the updated state.*

Solution 12.2: A straightforward hash function is to treat the 4×64 bits that constitute the board as a sequence of 64 digits in base 13, and use the hash function $\sum_{i=0}^{63} c_i p^i$, where c_i is the digit in location i, and p is a prime (see Solution 12.1 on the facing page).

This hash function does have ability to be updated incrementally—if, for example, a black knight on one square is replaced by a white bishop, the hash code update simply requires subtracting $c_k^B p^{i_1}$ and $c_b^W p^{i_2}$, and adding $c_b^W p^{i_1}$ and $c_\emptyset p^{i_2}$, where i_1 and i_2 are the initial locations of the knight and the bishop, respectively, and $c_k^B, c_b^W, c_\emptyset$ are the codes for black knight, white bishop, and empty space, respectively.

A more efficient hash function is based on creating a random 64-bit integer code for each of the 4×64 assignments of pieces to squares. The hash code for the state of the chessboard is the XOR of the code for each piece. Updates are trivial—for the example above, we XOR the code for black knight on i_1, white bishop on i_2, white bishop on i_1, and blank on i_2.

Problem 12.3, *pg. 93*: *Let s be an array of strings. Write a function which finds a closest pair of equal entries. For example, if s = ["All", "work", "and", "no", "play", "makes", "for", "no", "work", "no", "fun", "and", "no", "results"], then the second and third occurrences of "no" is the closest pair.*

Solution 12.3: We make a scan through the array. For each i, we determine the index j of the most recent occurrence of $s[i]$. If $i - j$ is less than the difference of the closest duplicate pair seen so far, we update that difference to $i - j$. The most recent occurrence of $s[i]$ is computed through a hash table lookup. The time complexity is $O(n)$, since we perform a constant amount of work per entry. The space complexity is $O(d)$, where d is the number of distinct strings in the array.

```
int find_nearest_repetition(const vector<string> &s) {
  unordered_map<string, int> string_to_location;
  int closest_dis = numeric_limits<int>::max();
  for (int i = 0; i < s.size(); ++i) {
    auto it = string_to_location.find(s[i]);
    if (it != string_to_location.end()) {
      closest_dis = min(closest_dis, i - it->second);
    }
    string_to_location[s[i]] = i;
  }
  return closest_dis;
}
```

Problem 12.4, *pg. 94*: *Given a set of binary trees $A_1, \ldots, A_n$ how would you compute a new set of binary trees $B_1, \ldots, B_n$ such that for each i, $1 \leq i \leq n$, A_i and B_i are isomorphic, and no pair of isomorphic nodes exists in the set of nodes defined by $B_1, \ldots, B_n$. (This is sometimes referred to as the canonical form.) Assume nodes are not shared in $A_1, \ldots, A_n$. See Figure 12.2 on Page 94 for an example.*

Solution 12.4: We will refer to $B_1, \ldots, B_n$ as the *canonical form* for $A_1, \ldots, A_n$. We can greatly accelerate the computation of the canonical form by caching. Specifically, we will cache the hash code for canonical nodes. Also, to compute the canonical node for a node n in some A_i, we will first compute the canonical nodes for n's children.

We need to define a hash function and an equality function for nodes. The hash function must have the property that isomorphic nodes are mapped to identical hash codes. The equality function should implement the isomorphism check.

The equality function can be implemented directly from the definition of isomorphism. The hash function can also be implemented fairly easily, e.g., $h(\texttt{null}) = 1$ and $h(x) = 3h(x.key) + 5h(x.left) + 7h(x.right)$.

```
static class BinaryTreeNode {
  int key;
  BinaryTreeNode left, right;
  Integer cachedHash;

  public BinaryTreeNode(int k, BinaryTreeNode l, BinaryTreeNode r) {
    this.key = k;
    this.left = l;
    this.right = r;
    this.cachedHash = null;
  }

  @Override
  public int hashCode() {
    if (this.cachedHash != null) {
      return this.cachedHash;
    }

    int x = 3 * key;
    int y = this.left == null ? 5 : 5 * this.left.hashCode();
    int z = this.right == null ? 7 : 7 * this.right.hashCode();
    this.cachedHash = x + y + z;
    return this.cachedHash;
  }

  @Override
  public boolean equals(Object o) {
    if (o == this) {
      return true;
    }
    if (!(o instanceof BinaryTreeNode)) {
      return false;
    }
    BinaryTreeNode n = (BinaryTreeNode)o;

```

```
    if (n == null || key != n.key) {
      return false;
    }
    // Assuming that equals is called on nodes
    // where children are already in canonical form
    return (left == n.left && right == n.right);
  }
}

static Map<BinaryTreeNode, BinaryTreeNode> nodeToCanonicalNode =
  new HashMap<BinaryTreeNode, BinaryTreeNode>();

static BinaryTreeNode getCanonical(BinaryTreeNode n) {
  BinaryTreeNode lc = (n.left == null) ? null : getCanonical(n.left);
  BinaryTreeNode rc = (n.right == null) ? null : getCanonical(n.right);
  BinaryTreeNode nc = new BinaryTreeNode(n.key, lc, rc);

  if (nodeToCanonicalNode.containsKey(nc)) {
    return nodeToCanonicalNode.get(nc);
  }
  nodeToCanonicalNode.put(nc, nc);
  return nc;
}
```

Incidentally, the implementation above illustrates what is known as the *flyweight pattern*.

ϵ-Variant 12.4.1: Design an efficient algorithm that computes the largest subtree common to two binary trees.

Problem 12.5, *pg. 94*: *You are given a sequence of users where each user has a unique 32-bit integer key and a set of attributes specified as strings. When you read a user, you should pair that user with another previously read user with identical attributes who is currently unpaired, if such a user exists. If the user cannot be paired, you should keep him in the unpaired set. How would you implement this matching process efficiently?*

Solution 12.5: Each user is associated with a set of attributes and we need to find users associated with a given set of attributes quickly. A hash table would be a perfect solution here but we need a hash function over the set of attributes. If the number of attributes is small, we can represent a subset of attributes as a bit array, where each bit represents a specific attribute. Once we have a canonical representation for sets, then we can use any hash function for bit arrays.

If the set of possible attributes is large, a better way to canonically represent a subset of attributes is to sort the attributes. (Any arbitrary ordering of attributes will work.) We can represent the sorted sequence of attributes as a string by concatenating the individual elements, and use a hash function for strings.

Problem 12.6, *pg. 95*: *Solve Problem 12.5 on Page 94 when users are grouped based on having similar attributes. The similarity between two sets of attributes A and B is* $\frac{|A \cap B|}{|A \cup B|}$.

Solution 12.6: Grouping users based on similarity makes the problem significantly more difficult. *Min-hashing* is a common approach. Essentially, we construct a set of k independent hash functions, $h_1, h_2, \ldots, h_k$. Then for the set of attributes S of each user, we define

$$M_k(S) = \min_{a_i \in S} h_k(a_i).$$

If two sets of attributes S_1 and S_2 are similar, then the probability of $M_k(S_1) = M_k(S_2)$ is high, specifically it is $|S_1 \cap S_2|/|S_1 \cap S_2|$. We map each set of attributes S to the sequence $\langle M_1(S), M_2(S), \ldots, M_k(S)\rangle$. We can use one of two criterion for identifying similar users. The first is that users who have the same sequences are potentially similar. The second is that users who have any hash code in common are candidates for being similar. The first criterion will have a higher false negative rate, whereas the second will have a higher false positive rate. In either case the problem has been reduced to hashing. The parameter k can be varied to increase or decrease the likelihood of false negatives.

Problem 12.7, *pg. 95*: *Write a function that takes as input a dictionary of English words, and returns a partition of the dictionary into subsets of words that are all anagrams of each other.*

Solution 12.7: Given a string s, let sort(s) be the string consisting of the characters in s rearranged so that they appear in sorted order. Observe that x and y are anagrams iff sort(x) = sort(y). For example, sort("logarithmic") and sort("algorithmic") are both "acghiilmort". Therefore anagrams can be identified by adding sort(s) for each string s in the dictionary to a hash table.

```
void find_anagrams(const vector<string> &dictionary) {
  // Get the sorted string and then insert into hash table
  unordered_map<string, vector<string>> hash;
  for (const string &s : dictionary) {
    string sorted_str(s);
    // Use sorted string as the hash code
    sort(sorted_str.begin(), sorted_str.end());
    hash[sorted_str].emplace_back(s);
  }

  for (const pair<string, vector<string>> &p : hash) {
    // Multiple strings with the same hash code => anagrams
    if (p.second.size() >= 2) {
      // Output all strings
      copy(p.second.begin(), p.second.end(),
           ostream_iterator<string>(cout, " "));
      cout << endl;
    }
  }
}
```

Problem 12.8, *pg. 95*: *Write a program to test whether the letters forming a string s can be permuted to form a palindrome. For example, "edified" can be permuted to form "deified".*

Explore solutions that trade time for space.

Solution 12.8: If the string is of even length, a necessary and sufficient condition for it to be a palindrome is that each character in the string appear an even number of times. If the length is odd, all but one character should appear an even number of times. Both these cases are covered by testing that at most one character appears an odd number of times, which can be checked using a hash table mapping characters to frequencies.

```
bool can_string_be_a_palindrome(const string &s) {
  unordered_map<char, int> hash;
  // Insert each char into hash
  for_each(s.begin(), s.end(), [&hash](const char &c) { ++hash[c]; });

  // A string can be permuted as a palindrome if the number of odd time
  // chars <= 1
  int odd_count = 0;
  for (const pair<char, int> &p : hash) {
    if (p.second & 1 && ++odd_count > 1) {
      break;
    }
  }
  return odd_count <= 1;
}
```

When the character set is large, we can perform the check without additional storage by sorting the string—this can be done in $O(n \log n)$ time and $O(1)$ space. Then we make a pass through the string determining character frequencies. (This approach changes the string itself.)

```
bool can_string_be_a_palindrome(string s) {
  sort(s.begin(), s.end());
  int odd_count = 0, num_curr_char = 1;

  for (int i = 1; i < s.size() && odd_count <= 1; ++i) {
    if (s[i] != s[i - 1]) {
      if (num_curr_char & 1) {
        ++odd_count;
      }
      num_curr_char = 1;
    } else {
      ++num_curr_char;
    }
  }
  if (num_curr_char & 1) {
    ++odd_count;
  }

  // A string can be permuted as a palindrome if the number of odd time
  // chars <= 1
  return odd_count <= 1;
}
```

Problem 12.9, *pg. 95*: *You are required to write a method which takes an anonymous letter L and text from a magazine M. Your method is to return* `true` *iff L can be written using M, i.e., if a letter appears k times in L, it must appear at least k times in M.*

Solution 12.9: In the problem scenario, it is likely that the string encoding the magazine is much longer than the string encoding the anonymous letter. We build a hash table H_L for L, where each key is a character in the letter and its value is the number of times it appears in the letter. Consequently, we scan the magazine character-by-character. When processing c, if c appears in H_L, we reduce its frequency count by 1; we remove it from H_L when its count goes to zero. If H_L becomes empty, we return `true`. If it is nonempty when we get to the end of M, we return `false`.

```
bool anonymous_letter(const string &L, const string &M) {
  unordered_map<char, int> hash;
  // Insert all chars in L into a hash table
  for_each(L.begin(), L.end(), [&hash](const char &c) { ++hash[c]; });

  // Check chars in M that could cover chars in a hash table
  for (const char &c : M) {
    auto it = hash.find(c);
    if (it != hash.cend()) {
      if (--it->second == 0) {
        hash.erase(it);
        if (hash.empty() == true) {
          return true;
        }
      }
    }
  }
  // No entry in hash means L can be covered by M
  return hash.empty();
}
```

Remark: If the characters are coded in ASCII, we could do away with H_L and use a 256 entry integer array A, with $A[i]$ being set to the number of times the character i appears in the letter.

Problem 12.10, *pg. 95*: *Let P be a set of n points in the plane. Each point has integer coordinates. Design an efficient algorithm for computing a line that contains the maximum number of points in P.*

Solution 12.10: Every pair of distinct points defines a line. We can use a hash table H to map lines to the set of points in P that lie on them. (Each corresponding set of points itself could be stored using a hash table.)

There are $\frac{n(n-1)}{2}$ pairs of points, and for each pair we have to do a lookup in H, an insert into H if the defined line is not already in H, and two inserts into the corresponding set of points. The hash table operations are $O(1)$ time, leading to an $O(n^2)$ time bound for this part of the computation.

We finish by finding the line with the maximum number of points with a simple iteration through the hash table searching for the line with the most points in its

corresponding set.

The design of a hash function appropriate for lines is more challenging than it may seem at first. The equation of line through (x_1, y_1) and (x_2, y_2) is

$$y = \frac{y_2 - y_1}{x_2 - x_1}x + \frac{x_2 y_1 - x_1 y_2}{x_2 - x_1}.$$

One idea would be to compute a hash code from the slope and the y-intercept of this line as an ordered pair of doubles. Because of finite precision arithmetic, we may have three points that are collinear map to distinct buckets.

A more robust hash function treats the slope and the y-intercept as rationals. A rational is an ordered pair of integers: the numerator and the denominator. We need to bring the rational into a canonical form before applying the hash function. One canonical form is to make the denominator always nonnegative, and relatively prime to the numerator. Lines parallel to the y-axis are a special case. For such lines, we use the x-intercept in place of the y-intercept, and use $\frac{1}{0}$ as the slope.

```
class Point {
  public:
    int x, y;

    //Equal function for hash
    const bool operator==(const Point &that) const {
      return x == that.x && y == that.y;
    }
};

// Hash function for Point
class HashPoint {
  public:
    const size_t operator()(const Point &p) const {
      return hash<int>()(p.x) ^ hash<int>()(p.y);
    }
};

// Line function of two points, a and b, and the equation is
// y = x(b.y - a.y) / (b.x - a.x) + (b.x * a.y - a.x * b.y) / (b.x - a.x)
class Line {
  private:
    static pair<int, int> get_canonical_fractional(int a, int b) {
      int gcd = GCD(abs(a), abs(b));
      a /= gcd, b /= gcd;
      return b < 0 ? make_pair(-a, -b) : make_pair(a, b);
    }

  public:
    // Store the numerator and denominator pair of slope unless the line is
    // parallel to y-axis that we store 1/0
    pair<int, int> slope;
    // Store the numerator and denominator pair of the y-intercept unless
    // the line is parallel to y-axis that we store the x-intercept
    pair<int, int> intercept;
```

```

    Line(const Point &a, const Point &b) :
      slope(a.x != b.x ? get_canonical_fractional(b.y - a.y, b.x - a.x) :
            make_pair(1, 0)),
      intercept(a.x != b.x ?
                get_canonical_fractional(b.x * a.y - a.x * b.y, b.x - a.x) :
                make_pair(a.x, 1)) {}

    // Equal function for hash
    const bool operator==(const Line &that) const {
      return slope == that.slope && intercept == that.intercept;
    }
};

// Hash function for Line
class HashLine {
  public:
    const size_t operator()(const Line &l) const {
      return hash<int>()(l.slope.first) ^ hash<int>()(l.slope.second) ^
             hash<int>()(l.intercept.first) ^ hash<int>()(l.intercept.second);
    }
};

Line find_line_with_most_points(const vector<Point> &P) {
  // Add all possible lines into hash table
  unordered_map<Line, unordered_set<Point, HashPoint>, HashLine> table;
  for (int i = 0; i < P.size(); ++i) {
    for (int j = i + 1; j < P.size(); ++j) {
      Line l(P[i], P[j]);
      table[l].emplace(P[i]), table[l].emplace(P[j]);
    }
  }

  // Return the line with most points have passed
  return max_element(table.cbegin(), table.cend(),
    [](const pair<Line, unordered_set<Point, HashPoint>> &a,
       const pair<Line, unordered_set<Point, HashPoint>> &b) {
      return a.second.size() < b.second.size();
    })->first;
}
```

Problem 12.11, *pg. 96*: *You are reading a sequence of strings separated by white space. You are allowed to read the sequence twice. Devise an algorithm that uses $O(k)$ memory to identify the words that occur at least $\lceil \frac{n}{k} \rceil$ times, where n is the length of the sequence.*

Solution 12.11: This is essentially a generalization of Problem 11.19 on Page 91. Here instead of discarding two distinct words, we discard $k+1$ distinct words at any given time and we are guaranteed that all the words that occurred more than $\frac{1}{k}$ times the length of the sequence prior to discarding continue to appear more than $\frac{1}{k}$ times in the remaining sequence. To implement this strategy, we need a hash table of the current candidates. Here is the code in C++:

```
vector<string> Search_frequent_items(istringstream &sin, const int &k) {
  // Find the candidates which may occur >= n / k times
  string buf;
  unordered_map<string, int> hash;
  int n = 0;  // count the number of strings

  while (sin >> buf) {
    ++hash[buf], ++n;
    // Detecting k + 1 items in hash, at least one of them must have exactly 1
    // in it. We will discard those k + 1 items by 1 for each.
    if (hash.size() == k + 1) {
      auto it = hash.begin();
      while (it != hash.end()) {
        if (--(it->second) == 0) {
          hash.erase(it++);  // remove the empty entry
        } else {
          ++it;
        }
      }
    }
  }

  // Reset hash for the following counting
  for (auto &it : hash) {
    it.second = 0;
  }

  // Reset the stream and read it again
  sin.clear();
  sin.seekg(0, ios::beg);
  // Count the occurrence of each candidate word
  while (sin >> buf) {
    auto it = hash.find(buf);
    if (it != hash.end()) {
      ++it->second;
    }
  }

  vector<string> ret;
  for (const pair<string, int> &it : hash) {
    // Select the word which occurs >= n / k times
    if (it.second >= static_cast<double>(n) / k) {
      ret.emplace_back(it.first);
    }
  }
  return ret;
}
```

The code may appear to take $O(nk)$ time since the inner loop may take k steps (decrementing count for all k entries) and the outer loop is called n times. However each word in the sequence can be erased only once, so the total time spent erasing is $O(n)$ and the rest of the steps inside the outer loop run in $O(1)$ time.

The first step yields a set S of not more than k words; set S is a superset of the words that occur greater than or equal to $\lceil \frac{n}{k} \rceil$ times. To get the exact set, we need to

make another pass over the sequence and count the number of times each word in S actually occurs. We return the words in S which occur greater than or equal to $\lceil \frac{n}{k} \rceil$ times.

Problem 12.12, *pg. 96*: *Design a scheme for checking membership in E that minimizes the number of disk accesses. Assume that* $|R| = 10^6, |E| = 10^5$, *and you can use up to* 1.25×10^5 *bytes of RAM.*

Solution 12.12: One approach may be to use the RAM to store E. However, we have only 1.125 bytes per word, which is not enough to store E. If we keep a subset of E in RAM, we cannot decide if a word is in R by looking at the subset of E. Alternately, we could keep a subset of R in RAM. However, R is 10× the size of E, and only a very small subset of R would fit. Potentially, we could sort R in descending order of frequency of occurrence, and fit as much as possible in RAM, but this is not effective if words occur uniformly.

A better approach is to use a *Bloom filter* to over-approximate E. We use the RAM to create a bit array A consisting of $n = 10^6$ bits, initialized to 0. We then have k hash functions $h_0, \ldots, h_{k-1}$ that we apply to each word w in E, and set bits $h_0(w) \bmod n, h_1(w) \bmod n, \ldots, h_{k-1}(w) \bmod n$ in A. When we get a new word x, we examine locations $h_0(x) \bmod n, h_1(x) \bmod n, \ldots, h_{k-1}(x) \bmod n$ in A—if any location is 0, we know $x \notin E$ and we use the rules to hyphenate it.

There may be words $x \in R$ for which locations $h_0(x) \bmod n, h_1(x) \bmod n, \ldots, h_{k-1}(x) \bmod n$ are all set to 1; however, when we go to disk we'll see that $x \notin E$, so we simply use the rules.

Since most words are in R and not in E, the intent is that A can be used to eliminate those words from R. The optimum choice for k depends on $|E|$ and n, and can be computed using simulation, or through analytical methods. It can be shown that in the optimum case, half the bits end up being set.

Problem 12.13, *pg. 96*: *A pair of strings is* k-suspicious *if they have a substring of length greater than or equal to k in common. Design an efficient algorithm that takes as input a set of strings and positive integer k, and returns all pairs of strings that are k-suspicious. Assume that most pairs will not be k-suspicious.*

Solution 12.13: Let l_i be the length of the i-th string. For each string we can compute $l_i - k + 1$ hash codes, one for *each* k length substring.

We insert these hash codes in a hash table G, recording which string each code corresponds to, and the offset the corresponding substring appears at. A collision indicates that the two length-k substrings are potentially the same.

Since we are computing hash code for each k length substring, it is important for efficiency to have a hash function which can be updated incrementally when we delete one character and add another. Solution 12.1 on Page 276 describes such a hash function.

In addition, it is important to have many slots in G, so that collisions of unequal strings is rare. A total of $\sum_{i=1}^{|S|}(l_i - k + 1)$ strings are added to the hash table. If k is

small relative to the string length and G has significantly fewer slots than the total number of characters in the strings, then we are certain to have collisions.

If it is not essential to return an exact answer, we can save storage by only considering a subset of substrings, e.g., those whose hash codes have 0s in the last b bits. This means that on average we consider $\frac{1}{2^b}$ of the total set of substrings (assuming the hash function does a reasonable job of spreading keys).

As an alternative to storing all the hash codes for all the strings in a hash table, we can record the hash codes in a Bloom filter, as in Solution 12.12 on the preceding page. This results in huge storage efficiencies, since in a Bloom filter, half the bits are set to one in an optimum configuration, whereas in the hash table approach, we want to have many more slots than elements. The negative of the Bloom filter approach is that it does not tell us where the potentially equal substring is. Based on the problem size, and the number of expected matching strings, we could create Bloom filters for subsets that partition the set of all strings, and then match each string d against the subsets, essentially using the Bloom filter to eliminate subsets of strings at a time from comparison with d.

Problem 12.14, *pg. 96*: *Let A and Q be arrays of strings. Define the subarray $A[i:j]$ to* cover *Q if for all $k \in [0, |Q|-1]$, there exists $l \in [i, j]$, $Q[k] = A[l]$. Write a function that takes two arrays A and Q and computes a minimum length subarray $A[i:j]$ that covers Q. Suppose that A is presented in streaming fashion, i.e., elements are read one at a time, and you cannot read earlier entries. The array Q is much smaller, and can be stored in RAM. How would you modify your solution for this case?*

Solution 12.14: We keep two pointers, left and right, which mark a minimal subarray of A that contains all the words in Q. A hash table H maps strings in Q that appear in the subarray to their frequency. The left and right pointers are initialized to 0. While the subarray does not contain all of Q, which is checked by looking at the size of H, we advance the right pointer. As soon as Q is covered, we record the separation of left and right, and advance left until Q is not covered. Then we advance right till it is covered again. In this way we track all minimal subarrays of A that contain Q. Assuming A is longer than Q (if it is not, we can immediately return `false`), the complexity is $O(n)$, where n is the length of A, since for each of the two pointers we do constant work per advance, and each is advanced at most $n-1$ times.

```
pair<int, int> find_smallest_subarray_covering_subset(
    const vector<string> &A, const vector<string> &Q) {
  unordered_set<string> dict(Q.cbegin(), Q.cend());
  unordered_map<string, int> count_Q;
  int l = 0, r = 0;
  pair<int, int> res(-1, -1);
  while (r < static_cast<int>(A.size())) {
    // Keep moving r until it reaches end or count_Q has |Q| items
    while (r < static_cast<int>(A.size()) && count_Q.size() < Q.size()) {
      if (dict.find(A[r]) != dict.end()) {
        ++count_Q[A[r]];
      }
      ++r;
```

```
    }

    if (count_Q.size() == Q.size() &&
        ((res.first == -1 && res.second == -1) ||
         r - 1 - l < res.second - res.first)) {
      res = {l, r - 1};
    }

    // Keep moving l until it reaches end or count_Q has less |Q| items
    while (l < static_cast<int>(A.size()) && count_Q.size() == Q.size()) {
      if (dict.find(A[l]) != dict.end()) {
        auto it = count_Q.find(A[l]);
        if (--(it->second) == 0) {
          count_Q.erase(it);
          if ((res.first == -1 && res.second == -1) ||
              r - 1 - l < res.second - res.first) {
            res = {l, r - 1};
          }
        }
      }
      ++l;
    }
  }
  return res;
}
```

The disadvantage of this approach is that we need to keep the subarrays in memory. We can achieve a streaming algorithm by keeping track of latest occurrences of query keywords as we process A. We use a doubly linked list L to store the last occurrence (index) of each keyword in Q, and hash table H to map each keyword in Q to the corresponding node in L. Each time a word in Q is encountered, we remove its node from L (which we find by using H), create a new node which records the current index in A, and append the new node to the end of L. We also update H. By doing this, each keyword in L is ordered by its order in A; therefore, if L has $|Q|$ words (i.e., all keywords are shown) and the current index minus the index stored in the first node in L is less than current best, we update current best. The complexity is still $O(n)$.

```
pair<int, int> find_smallest_subarray_covering_subset(
    istringstream &sin, const vector<string> &Q) {
  list<int> loc;  // tracks the last occurrence (index) of each string in Q
  unordered_map<string, list<int>::iterator> dict;
  for (const string &s : Q) {
    dict.emplace(s, loc.end());
  }

  pair<int, int> res(-1, -1);
  int idx = 0;
  string s;
  while (sin >> s) {
    auto it = dict.find(s);
    if (it != dict.end()) {  // s is in Q
      if (it->second != loc.end()) {
```

```
        loc.erase(it->second);
      }
      loc.emplace_back(idx);
      it->second = --loc.end();
    }

    if (loc.size() == Q.size() &&  // found |Q| keywords
        ((res.first == -1 && res.second == -1) ||
         idx - loc.front() < res.second - res.first)) {
      res = {loc.front(), idx};
    }
    ++idx;
  }
  return res;
}
```

Variant 12.14.1: Given an array A, find a longest subarray $A[i : j]$ such that all elements in $A[i : j]$ are distinct.

Problem 12.15, *pg. 97*: *Write a function that takes two integer-valued arrays A and Q and computes a minimum length subarray $A[i : j]$ that sequentially covers Q. Assume all elements in Q are distinct.*

Solution 12.15: We solve this with a single pass over the elements of A. We maintain three data structures:

(1.) A hash map K which maps each element of Q to its index in Q, i.e., $K(Q[j]) = j$.
(2.) An array L which maps j to the index of $Q[j]$'s most recent occurrence in A.
(3.) An array D which maps j to the length of the shortest subarray of A that ends at $L[j]$ and sequentially covers subarray $Q[0 : j]$.

When processing $A[i]$, if $A[i] = Q[j]$, we set $L[j] = i$. The update for D is based on the observation that if $A[i] = Q[j]$ then the shortest subarray of A that ends at $A[i]$ and sequentially covers subarray $Q[0 : j]$ consists of the shortest subarray that sequentially covers the nearest previous $Q[j-1]$ (which is at index $L[j-1]$) together with all the indices in $[L[j-1]+1, i]$. We use K to get j from $Q[j]$. Therefore, we can write the following equation for D:

$$D[j] = i - L[j-1] + D[j-1]$$

Processing each entry of A entails no more than a constant number of lookups and updates, leading to an $O(|A|)$ time complexity. The additional space complexity is dominated by the three hash tables, i.e., $O(|Q|)$.

```
pair<int, int> find_smallest_sequentially_covering_subset(
  const vector<string> &A, const vector<string> &Q) {
  unordered_map<string, int> K;
  vector<int> L(Q.size(), -1), D(Q.size(), numeric_limits<int>::max());

  // Initialize K
  for (int i = 0; i < Q.size(); ++i) {
```

```
    K.emplace(Q[i], i);
  }

  pair<int, int> res(-1, A.size());  // default value
  for (int i = 0; i < A.size(); ++i) {
    auto it = K.find(A[i]);
    if (it != K.cend()) {
      if (it->second == 0) {  // first one, no predecessor
        D[0] = 1;  // base condition
      } else if (D[it->second - 1] != numeric_limits<int>::max()) {
        D[it->second] = i - L[it->second - 1] + D[it->second - 1];
      }
      L[it->second] = i;

      if (it->second == Q.size() - 1 &&
          D.back() < res.second - res.first + 1) {
        res = {i - D.back() + 1, i};
      }
    }
  }
  return res;
}
```

Problem 12.16, *pg. 97*: *Implement a cache for looking up prices of books identified by their ISBN. Use the Least Recently Used (LRU) strategy for cache eviction policy.*

Solution 12.16: We use a hash table to quickly lookup price. Keys are ISBNs. Along with each key, we store the price and the most recent time a lookup was done on that key. However it takes $O(n)$ time to find the LRU item in a hash table with this scheme, where n is the number of entries in the hash table.

One way to improve performance is to use lazy garbage collection, which amortizes the cost of removing the LRU ISBNs. To be concrete, let's say we want the cache to be of size n. We do not delete any entries from the hash table until it grows to $2n$ entries. At this point we iterate through the entire hash table, and find the median age of items. Subsequently we discard everything below the median. The worst-case time to delete becomes $\Theta(n)$ but it will happen at most once every n operations. Therefore the amortized cost of deletion is $O(1)$ at the cost of doubling the memory consumption.

An alternative is to maintain a separate queue of keys. In the hash table we store for each key a reference to its location in the queue. Each time an ISBN is looked up that is found in the hash table, it is moved to the front of the queue. (This requires us to use a linked list implementation of the queue, so that items in the middle of the queue can be moved to the head.) When the queue exceeds length n, each time a lookup is performed that is not found in the hash table, the result is placed in the hash table and at the head of the queue; the item at the tail of the queue is deleted from the queue and the hash table.

```
template <typename ISBNType, typename PriceType, size_t capacity>
class LRUCache {
```

```
  private:
    typedef unordered_map<
        ISBNType, pair<typename list<ISBNType>::iterator, PriceType>> Table;
    Table cache;
    list<ISBNType> data;

    // Move the most recent accessed item to the front
    void moveToFront(const ISBNType &isbn,
                     const typename Table::iterator &it) {
      data.erase(it->second.first);
      data.emplace_front(isbn);
      it->second.first = data.begin();
    }

  public:
    const bool lookup(const ISBNType &isbn, PriceType* price) {
      auto it = cache.find(isbn);
      if (it == cache.end()) {
        return false;
      }

      *price = it->second.second;
      moveToFront(isbn, it);
      return true;
    }

    void insert(const ISBNType &isbn, const PriceType &price) {
      auto it = cache.find(isbn);
      if (it != cache.end()) {
        moveToFront(isbn, it);
      } else {
        // Remove the least recently used
        if (cache.size() == capacity) {
          cache.erase(data.back());
          data.pop_back();
        }

        data.emplace_front(isbn);
        cache.emplace(isbn, data.begin(), price);
      }
    }

    const bool erase(const ISBNType &isbn) {
      auto it = cache.find(isbn);
      if (it == cache.end()) {
        return false;
      }

      data.erase(it->second.first);
      cache.erase(it);
      return true;
    }
};
```

Problem 13.1, *pg. 98*: *What is the most efficient sorting algorithm for each of the following situations:*

- *A large array whose entries are random numbers.*
- *A small array of numbers.*
- *A large array of numbers that is already almost sorted.*
- *A large collection of integers that are drawn from a small range.*
- *A large collection of numbers most of which are duplicates.*
- *Stability is required, i.e., the relative order of two records that have the same sorting key should not be changed.*

Solution 13.1: In general, quicksort is considered one of the most efficient sorting algorithms since it has an average case run time of $\Theta(n \log n)$ and it sorts in-place (sorted data are not copied to some other buffer). For a large set of random integers, quicksort would be our choice.

Quicksort is more nuanced than appears at a first glance. For example, in a naïve implementation, an array with many duplicate elements leads to quadratic run times (and a high likelihood of stack space being exhausted because of the number of recursive calls). This can be managed by putting all keys equal to the pivot in the correct place. Similarly, it is important to call the smaller subproblem first—this, in conjunction with tail recursion ensures that the stack depth is $O(\log n)$.

However there are cases where other sorting algorithms are preferable.

- Small set—for a small set (for example, 3–10 numbers), a simple implementation such as insertion sort is easier to code, and runs faster in practice.
- Almost sorted array—if every element is known to be at most k places from its final location, a min-heap can be used to get an $O(n \log k)$ algorithm (Solution 10.6 on Page 253).
- Integers from a small range, or a few distinct keys—counting sort, which records for each element, the number of elements less than it. This count can be kept in an array (if the largest number is comparable in value to the size of the set being sorted) or a BST, where the keys are the numbers and the values are their frequencies.
- Many duplicates—we can add the keys to a BST, with linked lists for elements which have the same key; the sorted result can be derived from an in-order walk of the BST
- Stability is required—most useful sorting algorithms are not stable. Merge sort, carefully implemented, can be made stable; another solution is to add the index as an integer rank to the keys to break ties.

Problem 13.2, *pg. 99*: *Sort lines of a text file that has one million lines such that the average length of a line is* 100 *characters but the longest line is one million characters long.*

Solution 13.2: Almost all sorting algorithms rely on swapping records. However this becomes complicated when the record size varies. One way of dealing with this problem is to allocate for the maximum possible size for each record—this can be wasteful if there is a large variation in the sizes.

The better solution is *indirect sort*. First, build an array P of pointers to the records. Then sort the pointers using the compare function on the dereferenced pointers. Finally, iterate through P writing the dereferenced pointers.

```
template <typename T>
void indirect_sort(const string &file_name) {
  // Store file records into A
  ifstream ifs(file_name.c_str());
  vector<T> A;
  T x;
  while (ifs >> x) {
    A.emplace_back(x);
  }

  // Initialize P
  vector<T*> P;
  for (T &a : A) {
    P.emplace_back(&a);
  }

  // Indirect sort file
  sort(P.begin(), P.end(), [](const T* a, const T* b) -> bool {
                              return *a < *b;
                            });

  // Output file
  ofstream ofs(file_name.c_str());
  for (const T* p : P) {
    ofs << p << endl;
  }
}
```

Problem 13.3, *pg. 99*: *Design a sorting algorithm that minimizes the total distance that items are moved.*

Solution 13.3: Whenever the swap operation for the objects being sorted is expensive, one of the best things to do is indirect sort, i.e., sort references to the objects first and then apply the permutation that was applied to the references in the end.

In the case of statues, we can assign increasing indices to the statues from left-to-right and then sort the pairs of statue height and index. The indices in the sorted pairs would give us the permutation to apply. While applying permutation, we would want to perform it in a way that we move each statue the minimum possible distance. We can achieve this if each statue is moved exactly to its correct destination exactly once (no intermediate swaps).

Problem 13.4, *pg. 99*: *You are given an array of n* `Person` *objects. Each* `Person` *object has a field* `key`*. Rearrange the elements of the array so that* `Person` *objects with equal keys appear together. The order in which distinct keys appear is not important. Your algorithm must run in $O(n)$ time and $O(k)$ additional space. How would your solution change if keys have to appear in sorted order?*

Solution 13.4: We use the approach described in the introduction to the problem. However, we cannot apply it directly, since we need to write objects, not integers—two objects may have the same key but other fields may be different.

We use a hash table C to count the number of distinct occurrences of each key. We iterate over each key k in C and keep a cumulative count s which is the starting offset in the array where elements with key k are to be placed. We put the key-value pair (k, s) in a hash table M—basically M partitions the array into the subarrays holding objects with equal keys.

We then iteratively get a key k from M and swap the element e at k's current offset (which we get from M) with the location appropriate for e's key *e.key* (which we also get from M). Since e is now in its correct location, we update M by advancing the offset corresponding to *e.key*, taking care to remove *e.key* from M when all elements with key equal to *e.key* are correctly placed.

The time complexity is $O(n)$, since the first pass entails n hash table inserts, and the second pass performs a constant amount of work to move one element to the right location. (Selecting an arbitrary key from a hash table is a constant time operation.) The additional space complexity dictated by C and M, and is $O(k)$, where k is the number of distinct keys.

If the objects are additionally required to appear in sorted key order, we can store M using a BST-based map instead of a hash table. The time complexity becomes $O(n + k \log k)$, since BST insertion takes time $O(\log k)$. This should make sense, since if $k = n$, we are doing a regular sort, which is known to be $O(n \log n)$ for sorting based on comparisons.

```
template <typename KeyType>
void counting_sort(vector<Person<KeyType>> &people) {
  unordered_map<KeyType, int> key_to_count;
  for (const Person<KeyType> &p : people) {
    ++key_to_count[p.key_];
  }
  unordered_map<KeyType, int> key_to_offset;
  int offset = 0;
  for (const auto p : key_to_count) {
    key_to_offset[p.first] = offset;
    offset += p.second;
  }

  while (key_to_offset.size()) {
    auto from = key_to_offset.begin();
    auto to = key_to_offset.find(people[from->second].key_);
    swap(people[from->second], people[to->second]);
    // Use key_to_count to see when we are finished with a particular key
    if (--key_to_count[to->first]) {
      ++to->second;
    } else {
      key_to_offset.erase(to);
    }
  }
}
```

Problem 13.5, *pg. 99*: *Given sorted arrays A and B of lengths n and m respectively, return an array C containing elements common to A and B. The array C should be free of duplicates. How would you perform this intersection if—(1.) $n \approx m$ and (2.) $n \ll m$?*

Solution 13.5: The simplest algorithm is a "loop join", i.e., walking through all the elements of one array and comparing them to the elements of the other array. This has $O(mn)$ time complexity, regardless of whether the arrays are sorted or unsorted:

```
template <typename T>
vector<T> intersect_arrs1(const vector<T> &A, const vector<T> &B) {
  vector<T> intersect;
  for (int i = 0; i < A.size(); ++i) {
    if (i == 0 || A[i] != A[i - 1]) {
      for (int j = 0; j < B.size(); ++j) {
        if (A[i] == B[j]) {
          intersect.emplace_back(A[i]);
          break;
        }
      }
    }
  }
  return intersect;
}
```

However since both the arrays are sorted, we can make some optimizations. First, we can scan array A and use binary search in array B, find whether the element is present in B.

```
template <typename T>
vector<T> intersect_arrs2(const vector<T> &A, const vector<T> &B) {
  vector<T> intersect;
  for (int i = 0; i < A.size(); ++i) {
    if ((i == 0 || A[i] != A[i - 1]) &&
        binary_search(B.cbegin(), B.cend(), A[i])) {
      intersect.emplace_back(A[i]);
    }
  }
  return intersect;
}
```

Now our algorithm time complexity is $O(n \log m)$. We can further improve our run time by choosing the longer array for the inner loop since if $n \ll m$ then $m \log(n) \gg n \log(m)$.

This is the best solution if one set is much smaller than the other. However it is not optimal for cases where the set sizes are similar because we are not using the fact that both arrays are sorted to our advantage. In that case, iterating in tandem through the elements of each array in increasing order will work best as shown in this C++ code:

```
template <typename T>
vector<T> intersect_arrs3(const vector<T> &A, const vector<T> &B) {
  vector<T> intersect;
  int i = 0, j = 0;
```

```
  while (i < A.size() && j < B.size()) {
    if (A[i] == B[j] && (i == 0 || A[i] != A[i - 1])) {
      intersect.emplace_back(A[i]);
      ++i, ++j;
    } else if (A[i] < B[j]) {
      ++i;
    } else {  // A[i] > B[j]
      ++j;
    }
  }
  return intersect;
}
```

The run time for this algorithm is $O(m + n)$.

Problem 13.6, *pg. 100*: *Design an algorithm that takes as input two teams and the heights of the players in the teams and checks if it is possible to place players to take the photo subject to the placement constraint.*

Solution 13.6: Let A and B be arrays. Write $A < B$ if $A[i] < B[i]$ for each i. Let $\langle x_0, x_1, \ldots, x_{n-1} \rangle$ be an array of the heights of the players in Team X and $\langle y_0, y_1, \ldots, y_{n-1} \rangle$ be an array of the heights of the players in Team Y. By the transitivity of $<$, Team X can be placed in front of Team Y iff $\text{sort}\langle x_0, \ldots, x_{n-1} \rangle < \text{sort}\langle y_0, \ldots, y_{n-1} \rangle$, so the problem reduces to sorting. Figure 21.9 shows the teams in Figure 13.1 on Page 100 sorted by their heights.

Figure 21.9: The teams from Figure 13.1 on Page 100 in sorted order.

```
template <typename HeightType>
class Player {
  public:
    HeightType height;

    const bool operator<(const Player &that) const {
      return height < that.height;
    }
};

template <typename HeightType>
class Team {
  private:
    vector<Player<HeightType>> members;

    vector<Player<HeightType>> sortHeightMembers(void) const {
      vector<Player<HeightType>> sorted_members(members);
      sort(sorted_members.begin(), sorted_members.end());
      return sorted_members;
    }
```

```

  public:
    const bool operator<(const Team &that) const {
      vector<Player<HeightType>> this_sorted(sortHeightMembers());
      vector<Player<HeightType>> that_sorted(that.sortHeightMembers());
      for (int i = 0; i < this_sorted.size() && i < that_sorted.size(); ++i) {
        if (this_sorted[i] < that_sorted[i] == false) {
          return false;
        }
      }
      return true;
    }
};
```

Problem 13.7, *pg. 100*: *Given a string s, print in alphabetical order each character that appears in s, and the number of times that it appears. For example, if s ="bcdacebe", output "(a, 1), (b, 2), (c, 2), (d, 1), (e, 2)"..*

Solution 13.7: Many solutions exist for this problem. In the code below, we treat the string as an array of characters and sort that array; consequently, we iterate through the sorted array and count the number of occurrences of each character.

```
void count_occurrences(string S) {
  sort(S.begin(), S.end());

  int count = 1;
  for (int i = 1; i < S.size(); ++i) {
    if (S[i] == S[i - 1]) {
      ++count;
    } else {
      cout << '(' << S[i - 1] << ',' << count << ")";
      count = 1;
    }
  }
  cout << '(' << S.back() << ',' << count << ')' << endl;;
}
```

As an alternative, we could use an auxiliary array of integers indexed by characters, which counts the number of occurrences of each character. The sort routine itself can be based on radix sort, since the elements can be viewed as integers taking values in $[0, M-1]$ for a relatively small M. (We could also use a BST or a hash table in which keys are characters, and values are counts.)

Variant 13.7.1: The array A is an array of person objects. A person object has an age field, which is an integer in the range $[0, 150]$, and a name field, which is a string. The array is to be sorted on the age field and the sort must be stable. Design an $O(n)$ time algorithm for sorting A. Is it possible to sort in $O(n)$ time if ties are to be broken on the name field?

Problem 13.8, *pg. 100*: *Design an efficient algorithm for removing all the duplicates from an array.*

Solution 13.8: An efficient way of eliminating duplicates from any set of records, where a "less-than" operator can be defined, is to sort the records and then eliminate the duplicates in a single pass over the data.

Sorting can be done in $O(n \log n)$ time; the subsequent elimination of duplicates takes $\Theta(n)$ time. If the elimination of duplicates is done in-place, it would be more efficient than writing the unique set in a separate array since we would achieve better cache performance. Here is the code that does in-place duplicate removal:

```
template <typename T>
int eliminate_duplicate(vector<T> &A) {
  sort(A.begin(), A.end());  // makes identical elements become neighbors
  auto it = unique(A.begin(), A.end());  // removes neighboring duplicates
  A.resize(it - A.cbegin());  // truncates the unnecessary trailing part
  return it - A.cbegin();
}
```

Another efficient way is to use hash table where we store each record into a hash table as the key with no value and then write out all the keys in the hash table. Since hash table inserts can be done in $O(1)$ time and iterating over all the keys takes $O(n)$ time, this solution has much better time complexity than the sorting approach. However, in practice for small inputs, the sorting approach will likely be faster since it can be done in-place.

A counting sort (Solution 13.4 on Page 293 is appropriate if the number of distinct elements is small and the array is very large.

ϵ-Variant 13.8.1: Given an array A with possible duplicate entries, find the k entries that occur most frequently.

Problem 13.9, *pg. 101*: *Design an efficient algorithm that takes as input an array A of even length and computes a 2-partition of A that has minimum $Q(\Pi)$.*

Solution 13.9: Consider the task l that has the longest duration, i.e., $A[l] \geq A[i]$ for all $i \neq l$. Task l must be assigned to some worker w, and that worker must do some other additional Task s. The fastest w can complete both tasks is $A[l] + A[s]$, and this is minimized for sum_w such that $A[s]$ is a minimum element of A.

We now claim that there is always an optimum assignment in which the longest duration Task l and shortest duration Task s are assigned to the same worker.

Proof:

Let α be any assignment in which l and s are not paired. Suppose α pairs l with s' and s with l', where by supposition $A[l] \geq A[l']$ and $A[s] \leq A[s']$. Consider the assignment β where l and s are paired and l' and s' are paired, with the remaining pairings unchanged from α. Observe $A[l] + A[s'] \geq A[l] + A[s]$ due to $A[s'] \geq A[s]$. Since $A[l] \geq A[l']$ we have $A[l]+A[s'] \geq A[l']+A[s']$, which implies that the pairing of l' and s' in β is still better than the maximum delay pairing in α (which must

have duration at least $A[l] + A[s']$ or $A[l'] + A[s]$).

Note that we are not claiming that the time taken for the optimum assignment is $A[l]+A[s]$. Indeed this may not even be the case, e.g., if the two longest duration tasks are close to each other in duration and the shortest duration task takes much less time than the second shortest task. As a concrete example consider $A = \langle 1, 8, 9, 10 \rangle$ where maximum delay is $8 + 9 = 17$.

```
template <typename T>
vector<pair<T, T>> task_assignment(vector<T> A) {
  sort(A.begin(), A.end());
  vector<pair<T, T>> P;
  for (int i = 0, j = A.size() - 1; i < j; ++i, --j) {
    P.emplace_back(A[i], A[j]);
  }
  return P;
}
```

Problem 13.10, *pg. 101*: *Given a set of events, how would you determine the maximum number of events that take place concurrently?*

Solution 13.10: Each event corresponds to an interval $[b, e]$; let b and e be the earliest starting time and last ending time. Define the function $c(t)$ for $t \in [b, e]$ to be the number of intervals containing t. Observe that $c(\tau)$ does not change if τ is not the starting or ending time of an event.

This leads to an $O(n^2)$ brute-force algorithm, where n is the number of intervals: for each interval, for each of its two endpoints, determining how many intervals contain that point. The total number of endpoints is $2n$ and each check takes $O(n)$ time, since checking whether an interval $[b_i, e_i]$ contains a point t takes $O(1)$ time (simply check if $b_i \leq t \leq e_i$).

We can improve the run time to $O(n \log n)$ by sorting the set of all the endpoints in ascending order. If two endpoints have equal times, and one is a start time and the other is an end time, the one corresponding to a start time comes first. (If both are start or finish times, we break ties arbitrarily.)

We initialize a counter to 0, and iterate through the sorted sequence S from smallest to largest. For each endpoint that is the start of an interval, we increment the counter by 1, and for each endpoint that is the end of an interval, we decrement the counter by 1. The maximum value attained by the counter is maximum number of overlapping intervals.

```
template <typename TimeType>
class Interval {
  public:
    TimeType start, finish;
};

template <typename TimeType>
class Endpoint {
  public:
```

```
    TimeType time;
    bool isStart;

    const bool operator<(const Endpoint &e) const {
      return time != e.time ? time < e.time : (isStart && !e.isStart);
    }
};

template <typename TimeType>
int find_max_concurrent_events(const vector<Interval<TimeType>> &A) {
  // Build the endpoint array
  vector<Endpoint<TimeType>> E;
  for (const Interval<TimeType> &i : A) {
    E.emplace_back(Endpoint<TimeType>{i.start, true});
    E.emplace_back(Endpoint<TimeType>{i.finish, false});
  }
  // Sort the endpoint array according to the time
  sort(E.begin(), E.end());

  // Find the maximum number of events overlapped
  int max_count = 0, count = 0;
  for (const Endpoint<TimeType> &e : E) {
    if (e.isStart) {
      max_count = max(++count, max_count);
    } else {
      --count;
    }
  }
  return max_count;
}
```

ϵ-Variant 13.10.1: Users $1, 2, \ldots, n$ share an Internet connection. User i uses b_i bandwidth from time s_i to f_i, inclusive. What is the peak bandwidth usage?

Problem 13.11, *pg. 102*: *Design an algorithm that takes as input a set of intervals I, and outputs the union of the intervals. What is the time complexity of your algorithm as a function of the number of intervals?*

Solution 13.11: We begin with by sorting the intervals I on their left endpoints. If left endpoints a and b are equal, with a corresponding to a closed interval and b to an open interval, a comes first; otherwise, we break ties arbitrarily.

Let the sorted sequence be $\langle I_0, I_1, \ldots, I_{n-1} \rangle$. We create the result $\langle R_0, R_1, \ldots, R_m \rangle$ where $m \leq n$ by processing intervals in order; the R_is will be sorted by their left endpoints. Let t and s be interval-valued variables initialized to I_0, and I_1, respectively; we will show how to extend t to R_0. Let the left and right endpoints of $t(s)$ be $t.l(s.l)$ and $t.r(s.r)$, respectively. We have the following cases:

- $(s.l > t.r)$: R_0 is t, since no later interval can overlap or be adjacent to t.
- $(s.l = t.r)$ and (s is left open and t is right open): we set R_0 to t, since s and t cannot be merged and no later interval can overlap or be adjacent to t.

– ($s.l < t.r$) or ($s.l = t.r$ and (s is left-closed or t is right-closed)): if $s.r > t.r$ or ($s.r = t.r$ and s is right-closed) we extend t's right endpoint to $s.r$, and t is right open iff s is right open. We assign s to the next unprocessed interval and continue.

The code below implements this case analysis iteratively:

```
template <typename TimeType>
class Interval {
  private:
    class Endpoint {
      public:
        bool isClose;
        TimeType val;
    };

  public:
    Endpoint left, right;

    const bool operator<(const Interval &i) const {
      return left.val != i.left.val ?
             left.val < i.left.val : (left.isClose && !i.left.isClose);
    }
};

template <typename TimeType>
vector<Interval<TimeType>> Union_intervals(vector<Interval<TimeType>> I) {
  // Empty input
  if (I.empty()) {
    return {};
  }

  // Sort intervals according to their left endpoints
  sort(I.begin(), I.end());
  Interval<TimeType> curr(I.front());
  vector<Interval<TimeType>> uni;
  for (int i = 1; i < I.size(); ++i) {
    if (I[i].left.val < curr.right.val ||
        (I[i].left.val == curr.right.val &&
         (I[i].left.isClose || curr.right.isClose))) {
      if (I[i].right.val > curr.right.val ||
          (I[i].right.val == curr.right.val && I[i].right.isClose)) {
        curr.right = I[i].right;
      }
    } else {
      uni.emplace_back(curr);
      curr = I[i];
    }
  }
  uni.emplace_back(curr);
  return uni;
}
```

Problem 13.12, *pg. 102*: *You are given a set of n tasks modeled as closed intervals* $[a_i, b_i]$, *for*

$i = 0, \ldots, n-1$. *A set S of visit times* covers *the tasks if* $[a_i, b_i] \cap S \neq \emptyset$, *for* $i = 0, \ldots, n-1$. *Design an efficient algorithm for finding a minimum cardinality set of visit times that covers all the tasks.*

Solution 13.12: A covering set S must contain at least one point x such that $x \leq b_{min} = \min_{i=0}^{n-1} b_i$. Any such point covers the subset of intervals $[a_i, b_i], a_i \leq b_{min}$. Of course, b_{min} itself covers all such intervals and so there exists a minimum cardinality covering that contains b_{min} and no other points to its left. The same principle can be applied to the remaining intervals.

```
template <typename TimeType>
class Interval {
  public:
    TimeType left, right;
};

template <typename TimeType>
class LeftComp {
  public:
    const bool operator()(const Interval<TimeType> &a,
                          const Interval<TimeType> &b) const {
      return a.left != b.left ? a.left < b.left : a.right < b.right;
    }
};

template <typename TimeType>
class RightComp {
  public:
    const bool operator()(const Interval<TimeType> &a,
                          const Interval<TimeType> &b) const {
      return a.right != b.right ? a.right < b.right : a.left < b.left;
    }
};

template <typename TimeType>
vector<TimeType> find_minimum_visits(const vector<Interval<TimeType>> &I) {
  set<Interval<TimeType>, LeftComp<TimeType>> L;
  set<Interval<TimeType>, RightComp<TimeType>> R;
  for (const Interval<TimeType> &i : I) {
    L.emplace(i), R.emplace(i);
  }

  vector<TimeType> S;
  while (L.size() && R.size()) {
    TimeType b = R.cbegin()->right;
    S.emplace_back(R.cbegin()->right);

    // Remove the intervals which intersect with R.cbegin()
    auto it = L.cbegin();
    while (it != L.end() && it->left <= b) {
      R.erase(*it);
      L.erase(it++);
    }
  }
```

```
  return S;
}
```

Using a balanced BST (e.g., set in C++), we can implement the search for minimum, insertion, and deletion in $O(\log n)$ time, yielding an $O(n \log n)$ algorithm.

Problem 13.13, *pg. 102*: *Let $[\theta_i, \phi_i]$, for $i = 0, \ldots, n-1$ be n arcs, where the i-th arc is the set of points on the perimeter of the unit circle that subtend an angle in the interval $[\theta_i, \phi_i]$ at the center. A ray is a set of points that all subtend the same angle to the origin, and is identified by the angle they make relative to the x-axis. A set R of rays "covers" the arcs if $[\theta_i, \phi_i] \cap R \neq \emptyset$, for $i = 0, \ldots, n-1$. Design an efficient algorithm for finding a minimum cardinality set of rays that covers all arcs.*

Solution 13.13: If there exists a point on the circle that is not contained in at least one of the n arcs, the problem is identical to Problem 13.12 on Page 102. Therefore we assume every point is contained in one of the n arcs.

Without loss of generality, we may assume that a minimum cardinality covering set S contains only right endpoints of arcs, i.e., "clockwise" right endpoints. The total number of such endpoints is n. If we choose a given right endpoint and eliminate all the arcs that are covered by it, the remaining problem is identical to that in Problem 13.12 on Page 102. This means we can solve the arc-covering problem by n calls to the algorithm in Solution 13.12 on the facing page, yielding an $O(n^2 \log n)$ algorithm.

The approach of solving a problem involving a circular array by solving a number of instances of the same problem on linear arrays is fairly common—see for example Problem 15.5 on Page 118.

Problem 13.14, *pg. 103*: *Design an algorithm that takes as input an array A and a number t, and determines if A 3-creates t.*

Solution 13.14: We consider the case where $k = 2$ and A is sorted in Problem 11.4 on Page 87. Therefore, one solution is to sort A and for each $A[i]$, search for indices j and k such that $A[j] + A[k] = t - A[i]$. The additional space needed is $O(1)$, and the time complexity is the sum of the time taken to sort, $O(n \log n)$, and then to run the $O(n)$ algorithm in Solution 11.4 on Page 261 n times, which is $O(n^2)$ overall. The code for this approach is shown below.

```
template <typename T>
bool has_2_sum(const vector<T> &A, const T &t) {
  int j = 0, k = A.size() - 1;

  while (j <= k) {
    if (A[j] + A[k] == t) {
      return true;
    } else if (A[j] + A[k] < t) {
      ++j;
    } else {  // A[j] + A[k] > t
      --k;
    }
```

```
  }
  return false;
}

template <typename T>
bool has_3_sum(vector<T> A, const T &t) {
  sort(A.begin(), A.end());

  for (const T &a : A) {
    // Find if the sum of two numbers in A equals to t - a
    if (has_2_sum(A, t - a)) {
      return true;
    }
  }
  return false;
}
```

Remark: Surprisingly, it is possible, in theory, to improve the time complexity when the entries in A are nonnegative integers in a small range, specifically, the maximum entry is $O(n)$. The idea is to determine all possible 3-sums by encoding the array as a polynomial $P_A(x) = \sum_{i=0}^{n-1} x^{A[i]}$. The powers of x that appear in the polynomial $P_A(x) \times P_A(x)$ corresponds to sums of pairs of elements in A; similarly, the powers of x in $P_A(x) \times P_A(x) \times P_A(x)$ correspond to sums of triples of elements in A. Two n-degree polynomials can be multiplied in $O(n \log n)$ time using the fast Fourier Transform (FFT). The details are long and tedious, and the approach is unlikely to do well in practice.

ϵ-Variant 13.14.1: Solve the same problem when the three elements must be distinct. For example, if $A = [5, 2, 3, 4, 3]$ and $t = 9$, then $A[2] + A[2] + A[2]$ is not acceptable, $A[2] + A[2] + A[4]$ is not acceptable, but $A[1] + A[2] + A[3]$ and $A[1] + A[3] + A[4]$ are acceptable.

Variant 13.14.2: Solve the same problem when k is an additional input.

Problem 13.15, *pg. 103*: *Develop an algorithm for computing a short sequence of flips that will sort an array A.*

Solution 13.15: The most straightforward pancake sorting algorithm is analogous to insertion sort. Observe that we can move any pancake to any position with at most two flips: the first flip brings it to the top, the second flip moves it to the desired location. We can iteratively apply this idea to move the largest pancake to the bottom, followed by moving the second largest pancake to just above the largest pancake, etc. Note that the flips do not affect the positions of the pancakes that have been sorted by previous flips. Since we can put at least one pancake in the right place with two flips, this algorithm uses at most $2n$ flips. We can tighten this bound by observing that a stack of two pancakes requires at most one flip, so the number of flips made by the algorithm is actually $2(n-2) + 1 = 2n - 3$.

Note that we are trying to minimize the number of flips, not the time complexity of the algorithm that computes the sequence of flips. The run time of the algorithm is $O(n^2)$, since reversal takes $O(n)$ and finding the maximum element in an array also takes $O(n)$, and each of these operations could be performed once per pancake.

Remark: It is known that in the worst case $\frac{15}{14}n$ flips are necessary and $\frac{18}{11}n$ flips are sufficient. In other words, the straightforward algorithm does not always yield the optimum result. The first improvement was made by Bill Gates, when he was a math undergraduate at Harvard, in collaboration with Christos Papadimitriou.

Problem 14.1, *pg. 104*: *Write a function that takes as input the root of a binary tree whose nodes have a key field, and returns* `true` *iff the tree satisfies the BST property.*

Solution 14.1: Several solutions exist, which differ in terms of their space and time complexity, and the effort needed to code them.

The simplest is to start with the root r, and compute the maximum key *r.left.max* stored in the root's left subtree, and the minimum key *r.right.min* in the root's right subtree. Then we check that the key at the root is greater than or equal to *r.right.min* and less than or equal to *r.left.max*. If these checks pass, we continue checking the root's left and right subtree recursively.

Computing the minimum key in a binary tree is straightforward: we compare the key stored at the root with the minimum key stored in its left subtree and with the minimum key stored in its right subtree. The maximum key is computed similarly. (Note that the minimum may be in either subtree, since the tree may not satisfy the BST property.)

The problem with this approach is that it will repeatedly traverse subtrees. In a worst case, when the tree is BST and each node's left child is empty, its complexity is $O(n^2)$, where n is the number of nodes. The complexity can be improved to $O(n)$ by caching the largest and smallest keys at each node; this requires $O(n)$ additional storage.

We now present two approaches which have $O(n)$ time complexity and $O(h)$ additional space complexity.

The first, more straightforward approach, is to check constraints on the values for each subtree. The initial constraint comes from the root. Each node in its left (right) child must have a value less than or equal (greater than or equal) to the value at the root. This idea generalizes: if all nodes in a tree rooted at t must have values in the range $[l, u]$, and the value at t is $w \in [l, u]$, then all values in the left subtree of t must be in the range $[l, w]$, and all values stored in the right subtree of t must be in the range $[w, u]$. The code below uses this approach.

```
template <typename T>
bool is_BST_helper(const shared_ptr<BinaryTree<T>> &r, const T &lower,
                   const T &upper) {
  if (!r) {
    return true;
  } else if (r->data < lower || r->data > upper) {
    return false;
  }
```

```
  return is_BST_helper(r->left, lower, r->data) &&
         is_BST_helper(r->right, r->data, upper);
}

template <typename T>
bool is_BST(const shared_ptr<BinaryTree<T>> &r) {
  return is_BST_helper(r, numeric_limits<T>::min(), numeric_limits<T>::max());
}
```

The second approach is to perform an inorder traversal, and record the value stored at the last visited node. Each time a new node is visited, its value is compared with the value of the previous visited node; if at any step, the value at the previously visited node is greater than the node currently being visited, we have a violation of the BST property. In principle, this approach can use the existence of an $O(1)$ space complexity inorder traversal to further reduce the space complexity.

```
template <typename T>
bool is_BST(shared_ptr<BinaryTree<T>> n) {
  // Store the value of previous visited node
  int last = numeric_limits<T>::min();
  bool res = true;

  while (n) {
    if (n->left) {
      // Find the predecessor of n
      shared_ptr<BinaryTree<T>> pre = n->left;
      while (pre->right && pre->right != n) {
        pre = pre->right;
      }

      // Build the successor link
      if (pre->right) {  // pre->right == n
        // Revert the successor link if predecessor's successor is n
        pre->right = nullptr;
        if (last > n->data) {
          res = false;
        }
        last = n->data;
        n = n->right;
      } else {  // if predecessor's successor is not n
        pre->right = n;
        n = n->left;
      }
    } else {
      if (last > n->data) {
        res = false;
      }
      last = n->data;
      n = n->right;
    }
  }
  return res;
}
```

The approaches outlined above all explore the left subtree first. Therefore, even if the BST property does not hold at a node which is close to the root (e.g., the key stored at the right child is less than the key stored at the root), their time complexity is still $O(n)$.

We can search for violations of the BST property in a BFS manner to reduce the time complexity when the property is violated at a node whose depth is small, specifically much less than n.

The code below uses a queue to process nodes. Each queue entry contains a node, as well as an upper and a lower bound on the keys stored at the subtree rooted at that node. The queue is initialized to the root, with lower bound $-\infty$ and upper bound $+\infty$.

Suppose an entry with node n, lower bound l and upper bound u is popped. If n's left child is not `null`, a new entry consisting of *n.left*, upper bound *n.key* and lower bound l is added. A symmetric entry is added if n's right child is not `null`. When adding entries, we check that the node's key lies in the range specified by the lower bound and the upper bound; if not, we return immediately reporting a failure.

We claim that if the BST property is violated in the subtree consisting of nodes at depth d or less, it will be discovered without visiting any nodes at levels $d+1$ or more. This is because each time we enqueue an entry, the lower and upper bounds on the node's key are the tightest possible. A formal proof of this is by induction; intuitively, it is because we satisfy all the BST requirements induced by the search path to that node.

```
template <typename T>
class QNode {
  public:
    shared_ptr<BinaryTree<T>> node;
    T lower, upper;
};

template <typename T>
bool is_BST(const shared_ptr<BinaryTree<T>> &n) {
  queue<QNode<T>> q;

  q.emplace(QNode<T>{n, numeric_limits<T>::min(), numeric_limits<T>::max()});
  while (!q.empty()) {
    if (q.front().node) {
      if (q.front().node->data < q.front().lower ||
          q.front().node->data > q.front().upper) {
        return false;
      }

      q.emplace(QNode<T>{q.front().node->left, q.front().lower,
                         q.front().node->data});
      q.emplace(QNode<T>{q.front().node->right, q.front().node->data,
                         q.front().upper});
    }
    q.pop();
  }
  return true;
```

```
}
```

Problem 14.2, *pg. 105*: *Given a node x, find the successor of x in a BST. Assume that nodes have* `parent` *fields, and the* `parent` *field of root points to* `null`.

Solution 14.2: If there is a right subtree from the given node, we return the smallest node in the right subtree. If the node does not have a right child, then we need to keep going up the tree till we find a node which is the left child of its parent, in which case that parent is the desired successor. If we reach the root then the given node is the largest node in the tree and has no successor.

```
template <typename T>
shared_ptr<BinarySearchTree<T>> find_successor_BST(
  shared_ptr<BinarySearchTree<T>> n) {
  if (n->right) {
    // Find the smallest element in n's right subtree
    n = n->right;
    while (n->left) {
      n = n->left;
    }
    return n;
  }

  // Find the first parent which is larger than n
  while (n->parent && n->parent->right == n) {
    n = n->parent;
  }
  // Return nullptr means n is the largest in this BST
  return n->parent;
}
```

Problem 14.3, *pg. 105*: *Design efficient functions for inserting and removing keys in a BST. Assume that all elements in the BST are unique, and that your insertion method must preserve this property. You cannot change the contents of any node. What are the time complexities of your functions?*

Solution 14.3: Insertion is done as follows. We start by creating a new node c holding k. Observe that inserting a key k into a tree that is empty is trivial: we set c to be the root of the tree. If the tree is nonempty and k is greater than the key r stored at the root, we need to insert k into the root's right subtree. If r and k are equal, we return, since we do not want add a duplicate. Otherwise we insert into the root's left subtree. At some step, we will either return because we found a node containing a key equal to k, or encounter an empty tree. For this case, we need to set the appropriate child of the last node whose key we compared k with to c.

Deletion begins with first identifying the node d containing k. (If no such node exists, nothing needs to be done.) Suppose d has no right child and no left child. In this case, there is nothing to be done, beyond updating the appropriate child field in d's parent to `null`. Otherwise, if d has a right child, we find its successor, call it s,

which must lie in d's right subtree. We "substitute" d with s—this entails updating d's parent to point to s instead of d, and setting the children of s to the children of d. There are a couple of corner cases: if d has no parent, the parent update operation is skipped; if s is the right child of d, then the right child of s is set to `null`. If d does not have a right child, all we do is update d's parent to point to s. There is one corner case: if d has no parent, i.e., it is the root, the root is updated to s.

Both insertion and deletion times are dominated by the time taken to search for a key and to find the minimum element in a subtree. Both of these times are proportional to the height of the tree. In the worst case, the BST can grow to be very skewed. For example, if the initial tree is empty, and n successive insertions are done, where each key inserted is larger than the previous one, the height of the resulting tree is n. It is possible to modify the insertion and deletion routines to keep the tree height $O(\log n)$, where n is the number of nodes; furthermore, these insertion and deletion functions continue to have time complexity proportional to the height. AVL and red-black trees are BSTs with additional state related to height imbalance stored at each node. Specialized insertion and deletion operations do "rotations" about nodes to keep the height logarithmic in the number of nodes.

```
template <typename T>
class BinarySearchTree {
  private:
    class TreeNode {
      public:
        T data;
        shared_ptr<TreeNode> left, right;
    };

    void clear(shared_ptr<TreeNode> &n) {
      if (n) {
        clear(n->left), clear(n->right);
        n = nullptr;
      }
    }

    // Replace the link between par and child by new_link
    void replaceParentChildLink(shared_ptr<TreeNode> par,
                                shared_ptr<TreeNode> child,
                                shared_ptr<TreeNode> new_link) {
      if (!par) {
        return;
      }

      if (par->left == child) {
        par->left = new_link;
      } else {
        par->right = new_link;
      }
    }

    shared_ptr<TreeNode> root;

```

```
public:
  BinarySearchTree(void) : root(nullptr) {}

  ~BinarySearchTree(void) {
    clear();
  }

  const bool empty(void) const {
    return !root;
  }

  void clear(void) {
    clear(root);
  }

  const bool insert(const T &key) {
    shared_ptr<TreeNode>
      t = shared_ptr<TreeNode>(new TreeNode{key, nullptr, nullptr}),
      par = nullptr;

    if (empty()) {
      root = t;
    } else {
      shared_ptr<TreeNode> curr;
      curr = root;
      while (curr) {
        par = curr;
        if (t->data == curr->data) {
          t = nullptr;
          return false;  // no insertion for duplicate key
        } else if (t->data < curr->data) {
          curr = curr->left;
        } else {  // t->data > curr->data
          curr = curr->right;
        }
      }

      // Insert key according to key and par
      if (t->data < par->data) {
        par->left = t;
      } else {
        par->right = t;
      }
    }
    return true;
  }

  const bool erase(const T &key) {
    // Find the node with key
    shared_ptr<TreeNode> curr = root, par = nullptr;
    while (curr && curr->data != key) {
      par = curr;
      curr = key < curr->data ? curr->left : curr->right;
    }

```

```
      // No node with key in this binary tree
      if (!curr) {
        return false;
      }

      if (curr->right) {
        // Find the minimum of the right subtree
        shared_ptr<TreeNode> r_curr = curr->right, r_par = curr;
        while (r_curr->left) {
          r_par = r_curr;
          r_curr = r_curr->left;
        }
        // Move links to erase the node
        replaceParentChildLink(par, curr, r_curr);
        replaceParentChildLink(r_par, r_curr, r_curr->right);
        r_curr->left = curr->left, r_curr->right = curr->right;

        // Update root link if needed
        if (root == curr) {
          root = r_curr;
        }
      } else {
        // Update root link if needed
        if (root == curr) {
          root = curr->left;
        }
        replaceParentChildLink(par, curr, curr->left);
      }
      curr = nullptr;
      return true;
    }
};
```

Problem 14.4, *pg. 105*: *Given a BST T, write recursive and iterative versions of a function that takes a BST T, a key k, and returns the node containing k that would appear first in an inorder walk. If k is absent, return* `null`. *For example, when applied to the BST in Figure 14.2 on Page 106, your algorithm should return Node B if* $k = 108$, *Node G if* $k = 285$, *and* `null` *if* $k = 143$.

Solution 14.4: The standard way to search for a key k in a BST is to first check if the tree is empty; if so, we return `null`. Consequently check if the root stores k, in which case we return the root, and recur on the left/right child of the root, if k is less/greater than the value at the root.

For the problem we are given, we need to find the *first* occurrence of k that would appear in an inorder walk. We achieve this by making a simple modification to the standard search—if k is matched by the root, we also check to see if it appears in the root's left subtree, in which case we return the node returned by that call. Correctness follows from the fact that the nodes in the root's left subtree all appear before the root in an inorder walk.

```
template <typename T>
```

```
shared_ptr<BinarySearchTree<T>> find_first_equal_k(
    const shared_ptr<BinarySearchTree<T>> &r, const T &k) {
  if (!r) {
    return nullptr;  // no match
  } else if (r->data == k) {
    // Recursively search the left subtree for first one == k
    shared_ptr<BinarySearchTree<T>> n = find_first_equal_k(r->left, k);
    return n ? n : r;
  }
  // Search left or right tree according to r->data and k
  return find_first_equal_k(r->data < k ? r->right : r->left, k);
}
```

The straightforward implementation of the standard BST search is tail recursive, and can mechanically be converted to iterative code. However, the recursive version for finding the first occurrence of k is not tail recursive, and needs to be written from scratch. The iterative code below makes use of the elimination principle.

```
template <typename T>
shared_ptr<BinarySearchTree<T>> find_first_equal_k(
    shared_ptr<BinarySearchTree<T>> r, const T &k) {
  shared_ptr<BinarySearchTree<T>> first = nullptr;
  while (r) {
    if (r->data < k) {
      r = r->right;
    } else if (r->data > k) {
      r = r->left;
    } else {  // r->data == k
      // Search for the leftmost in the left subtree
      first = r;
      r = r->left;
    }
  }
  return first;
}
```

Problem 14.5, *pg. 106*: *Write a function that takes a BST T and a key k, and returns the first entry larger than k that would appear in an inorder walk. If k is absent or no key larger than k is present, return* `null`. *For example, when applied to the BST in Figure 14.1 on Page 105 you should return 29 if $k = 23$; if $k = 32$, you should return* `null`.

Solution 14.5: This problem can be solved using a reductionist approach. First we find the node n holding k that appears last in an inorder walk of the tree, and then we find n's successor. The node n can be found using a straightforward modification of Solution 14.4 on the preceding page. If n does not exist (k is not the value stored at any node in the tree) or n has no successor (i.e., it is the last node in the inorder walk), we return `null`.

A more direct approach is to maintain a candidate node, `first`. The node `first` is initialized to `null`. Now we look for k using the standard search idiom. If the current node's key is larger than k, we update `first` to the current node and continue the

search in the left subtree. If the current node's key is smaller than k, we search in the right subtree. If the current node's key is equal to k, we set a Boolean-valued `found_k` variable to `true`, and continue search in the current node's right subtree. When the current node becomes `null`, if `found_k` is `true` we return `first`, otherwise we return `null`. Correctness follows from the fact that after `first` is assigned within the loop, the desired result is within the tree rooted at `first`. The concept of this approach is similar with Solution 11.2 on Page 259.

```
template <typename T>
shared_ptr<BinarySearchTree<T>> find_first_larger_k_with_k_exist(
    shared_ptr<BinarySearchTree<T>> r, const T &k) {
  bool found_k = false;
  shared_ptr<BinarySearchTree<T>> first = nullptr;

  while (r) {
    if (r->data == k) {
      found_k = true;
      r = r->right;
    } else if (r->data > k) {
      first = r;
      r = r->left;
    } else {  // r->data < k
      r = r->right;
    }
  }
  return found_k ? first : nullptr;
}
```

Problem 14.6, *pg. 106*: *Write a function that takes a min-first BST T and a key k, and returns true iff T contains k.*

Solution 14.6: The algorithm proceeds in the following sequence:

1. If T is empty, or if k is less than the root, it cannot be present in T; we return false.
2. Otherwise, if k equals the key stored at T's root, we return true.
3. Otherwise, we recursively search both left and right subtrees and return true if and only if it is present in either.

```
template <typename T>
bool search_min_first_BST(const shared_ptr<BinarySearchTree<T>> &r,
                          const T &k) {
  if (!r || r->data > k) {
    return false;
  } else if (r->data == k) {
    return true;
  }
  return search_min_first_BST(r->left, k) ||
         search_min_first_BST(r->right, k);
}
```

Variant 14.6.1: Print the keys in a min-first BST in sorted order.

Variant 14.6.2: A max-first BST is defined analogously to the min-first BST, the difference being that the largest key is stored at the root. Design an algorithm that takes an n node min-BST and converts it to a max-BST in $O(n)$ time. Use as little additional space as possible.

Variant 14.6.3: Implement insert and delete functions for a min-first BST.

Problem 14.7, *pg. 107*: *How would you build a BST of minimum possible height from a sorted array A?*

Solution 14.7: Intuitively, we want the subtrees to be as balanced as possible. One way of achieving this is to make the element at entry $\lfloor \frac{n}{2} \rfloor$ the root, and recursively compute minimum height BSTs for the subarrays $A[0 : \lfloor \frac{n}{2} \rfloor - 1]$ and $A[\lfloor \frac{n}{2} \rfloor + 1 : n-1]$.

```
// Build BST based on subarray A[start : end - 1]
template <typename T>
shared_ptr<BinarySearchTree<T>> build_BST_from_sorted_array_helper(
    const vector<T> &A, const int &start, const int &end) {
  if (start < end) {
    int mid = start + ((end - start) >> 1);
    return shared_ptr<BinarySearchTree<T>>(new BinarySearchTree<T>{
        A[mid],
        build_BST_from_sorted_array_helper(A, start, mid),
        build_BST_from_sorted_array_helper(A, mid + 1, end)}
    );
  }
  return nullptr;
}

template <typename T>
shared_ptr<BinarySearchTree<T>> build_BST_from_sorted_array(
    const vector<T> &A) {
  return build_BST_from_sorted_array_helper(A, 0, A.size());
}
```

Problem 14.8, *pg. 107*: *Let L be a singly linked list of numbers, sorted in ascending order. Design an efficient algorithm that takes as input L, and builds a height-balanced BST on the entries in L. Your algorithm should run in $O(n)$ time, where n is the number of nodes in L. You cannot use dynamic memory allocation—reuse the nodes of L for the BST. You can update pointer fields, but cannot change node contents.*

Solution 14.8: The straightforward algorithm entails finding the midpoint m of the list, creating the root with the corresponding key, and recursing on the first half and the second half of the list. The time complexity satisfies the recurrence $T(n) = O(n) + 2T(\frac{n}{2})$—the $O(n)$ term comes from the traversal required to find the midpoint of the list. This solves to $T(n) = O(n \log n)$.

If the nodes were in an array A, we could index directly into A to obtain m, and the time complexity would satisfy $S(n) = O(1) + 2S(\frac{n}{2})$, which solves to $S(n) = O(n)$.

We can avoid the additional space required to convert the list to an array by first building the tree with empty keys, and then populating it by performing an inorder walk of the tree in conjunction with a traversal of the list.

Specifically, first we find the length n of the list, $O(n)$ operation. We then create the balanced tree recursively—create the root node, and a balanced left child on $L = \lfloor \frac{n}{2} \rfloor$, and a balanced right child on $R = n - L - 1$ nodes. This entails a $O(1)$ time spent per node, and is also an $O(1)$ operation. The inorder walk entails calling the next method of the list iterator per visited node, leading to an $O(n)$ complexity for the final population of the tree nodes.

```
// Build a BST from the (s + 1)-th to the e-th node in L
template <typename T>
shared_ptr<BinarySearchTree<T>> build_BST_from_sorted_doubly_list_helper(
    shared_ptr<node_t<T>> &L, const int &s, const int &e) {
  shared_ptr<BinarySearchTree<T>> curr = nullptr;
  if (s < e) {
    int m = s + ((e - s) >> 1);
    curr = shared_ptr<BinarySearchTree<T>>(new BinarySearchTree<T>);
    curr->left = build_BST_from_sorted_doubly_list_helper(L, s, m);
    curr->data = L->data;
    L = L->next;
    curr->right = build_BST_from_sorted_doubly_list_helper(L, m + 1, e);
  }
  return curr;
}

template <typename T>
shared_ptr<BinarySearchTree<T>> build_BST_from_sorted_doubly_list(
    shared_ptr<node_t<T>> L, const int &n) {
  return build_BST_from_sorted_doubly_list_helper(L, 0, n);
}
```

Problem 14.9, *pg. 107*: *Design an algorithm that takes as input a BST B and returns a sorted doubly linked list on the same elements. Your algorithm should not allocate any new nodes. The original BST does not have to be preserved; use its nodes as the nodes of the resulting list, as shown in Figure 14.4 on Page 107.*

Solution 14.9: Here is a recursive solution. Build a list out of left subtree, append the root to it, and then append the list from the right subtree. Since we do a constant amount of work per tree node, the time complexity is $O(n)$, where n is the number of nodes in the BST. The space complexity is $\Theta(h)$, where h is the height of the BST. The worst case is for a completely left-skewed tree, i.e., a tree in which no node has a right child—n activation records are pushed on the stack.

```
// Transform a BST into a circular sorted doubly linked list in-place,
// return the head of the list
template <typename T>
shared_ptr<BinarySearchTree<T>> BST_to_doubly_list(
    const shared_ptr<BinarySearchTree<T>> &n) {
  // Empty subtree
  if (!n) {
```

```
    return nullptr;
  }

  // Recursively build the list from left and right subtrees
  auto l_head(BST_to_doubly_list(n->left));
  auto r_head(BST_to_doubly_list(n->right));

  // Append n to the list from left subtree
  shared_ptr<BinarySearchTree<T>> l_tail = nullptr;
  if (l_head) {
    l_tail = l_head->left;
    l_tail->right = n;
    n->left = l_tail;
    l_tail = n;
  } else {
    l_head = l_tail = n;
  }

  // Append the list from right subtree to n
  shared_ptr<BinarySearchTree<T>> r_tail = nullptr;
  if (r_head) {
    r_tail = r_head->left;
    l_tail->right = r_head;
    r_head->left = l_tail;
  } else {
    r_tail = l_tail;
  }
  r_tail->right = l_head, l_head->left = r_tail;

  return l_head;
}
```

Problem 14.10, *pg. 108*: *Let A and B be BSTs. Design an algorithm that merges them in $O(n)$ time. You cannot use dynamic allocation. You do not need to preserve the original trees. You can update pointer fields, but cannot change the key stored in a node.*

Solution 14.10: Our solution builds on Solution 14.9 on the previous page and Solution 14.8 on Page 314. We convert each BST into a doubly linked list using Solution 14.9 on the previous page, which runs in $O(n)$ time and $O(n)$ space. These two lists can be merged in $O(n)$ time and $O(1)$ space, as described in Solution 7.1 on Page 207. The resulting list can be converted into a BST using Solution 14.8 on Page 107, which uses $O(n)$ time and $O(\log n)$ space. Each of the three sub-routines does not explicitly allocate memory.

```
template <typename T>
void append_node(shared_ptr<BinarySearchTree<T>> &head,
                 shared_ptr<BinarySearchTree<T>> &tail,
                 shared_ptr<BinarySearchTree<T>> &n) {
  if (head) {
    tail->right = n, n->left = tail;
  } else {
    head = n;
```

```
  }
  tail = n;
}

template <typename T>
void append_node_and_advance(shared_ptr<BinarySearchTree<T>> &head,
                             shared_ptr<BinarySearchTree<T>> &tail,
                             shared_ptr<BinarySearchTree<T>> &n) {
  append_node(head, tail, n);
  n = n->right;  // advance n
}

// Merge two sorted linked lists, return the head of list
template <typename T>
shared_ptr<BinarySearchTree<T>> merge_sorted_linked_lists(
    shared_ptr<BinarySearchTree<T>> A, shared_ptr<BinarySearchTree<T>> B) {
  shared_ptr<BinarySearchTree<T>> sorted_list = nullptr, tail = nullptr;

  while (A && B) {
    append_node_and_advance(sorted_list, tail, A->data < B->data ? A : B);
  }

  // Append the remaining of A
  if (A) {
    append_node(sorted_list, tail, A);
  }
  // Append the remaining of B
  if (B) {
    append_node(sorted_list, tail, B);
  }
  return sorted_list;
}

template <typename T>
shared_ptr<BinarySearchTree<T>> merge_BSTs(
    shared_ptr<BinarySearchTree<T>> A, shared_ptr<BinarySearchTree<T>> B) {
  // Transform BSTs A and B into sorted doubly lists
  A = BST_to_doubly_list(A), B = BST_to_doubly_list(B);
  A->left->right = B->left->right = nullptr;
  A->left = B->left = nullptr;
  int len_A = count_len(A), len_B = count_len(B);
  return build_BST_from_sorted_doubly_list(merge_sorted_linked_lists(A, B),
                                           len_A + len_B);
}
```

Problem 14.11, *pg. 108*: *Given the root of a BST and an integer k, design a function that finds the k largest elements in this BST. For example, if the input to your function is the BST in Figure 14.1 on Page 105 and $k = 3$, your function should return* $\langle 53, 47, 43 \rangle$.

Solution 14.11: We do a reverse inorder traversal of the tree: visit the right subtree, visit the root, then visit the left subtree. This results in nodes being visited in descending order. As soon as we visit k nodes, we can halt. Note that the time complexity of this approach is still $O(n)$ even if $k \ll n$, e.g., if the BST is of the form

of a list. The code below uses a vector to store the desired keys; as soon as the vector has k elements, we return.

```
template <typename T>
void find_k_largest_in_BST_helper(const shared_ptr<BinarySearchTree<T>> &r,
                                  const int &k, vector<T> &k_elements) {
  // Perform reverse inorder traversal
  if (r && k_elements.size() < k) {
    find_k_largest_in_BST_helper(r->right, k, k_elements);
    if (k_elements.size() < k) {
      k_elements.emplace_back(r->data);
      find_k_largest_in_BST_helper(r->left, k, k_elements);
    }
  }
}

template <typename T>
vector<T> find_k_largest_in_BST(const shared_ptr<BinarySearchTree<T>> &root,
                                const int &k) {
  vector<T> k_elements;
  find_k_largest_in_BST_helper(root, k, k_elements);
  return k_elements;
}
```

Problem 14.12, *pg. 109*: *Which traversal orders—inorder, preorder, and postorder—of a BST can be used to reconstruct the BST uniquely? Write a program that takes as input a sequence of node keys and computes the corresponding BST. Assume that all keys are unique.*

Solution 14.12: The sequence of node keys generated by an inorder traversal is not enough to reconstruct the tree. For example, the sequence $\langle 1, 2, 3 \rangle$ corresponds to five distinct BSTs as shown in Figure 21.10.

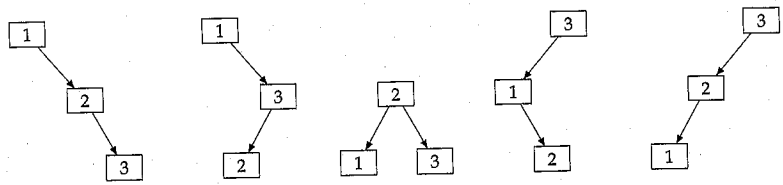


Figure 21.10: Five distinct BSTs for the traversal sequence $\langle 1, 2, 3 \rangle$.

We claim there exists a unique BST corresponding to a sequence of nodes visited in a preorder traversal.

Proof:

We use induction on n, the number of nodes. Only one tree exists on one node, so the base case holds. Assume the claim holds for all $n < k$. Consider a preorder traversal sequence σ of length k. The first value v in σ corresponds to the root. The subsequence σ_l of σ which begins at the second element of σ and ends at the last

value less than v, corresponds to the preorder traversal of the root's left subtree. Its length is less than k. Hence by induction we can reconstruct the left subtree uniquely. The right subtree corresponds to the subsequence following σ_l, and can also be reconstructed uniquely by induction.

It is critical that the elements stored in the tree be unique. If the root contains v and the tree contains more occurrences of v, we cannot identify from the sequence whether the subsequent vs are in the left subtree or the right subtree.

The above proof can be used to rebuild the BST from a preorder sequence.

```
// Build a BST based on preorder[s : e - 1], return its root
template <typename T>
shared_ptr<BinarySearchTree<T>> rebuild_BST_from_preorder_helper(
    const vector<T> &preorder, const int &s, const int &e) {
  if (s < e) {
    int x = s + 1;
    while (x < e && preorder[x] < preorder[s]) {
      ++x;
    }
    return shared_ptr<BinarySearchTree<T>>(new BinarySearchTree<T>{
        preorder[s],
        rebuild_BST_from_preorder_helper(preorder, s + 1, x),
        rebuild_BST_from_preorder_helper(preorder, x, e)});
  }
  return nullptr;
}

// Given a preorder traversal of a BST, return its root
template <typename T>
shared_ptr<BinarySearchTree<T>> rebuild_BST_from_preorder(
    const vector<T> &preorder) {
  return rebuild_BST_from_preorder_helper(preorder, 0, preorder.size());
}
```

The worst-case input for this algorithm is the pre-order sequence corresponding to a completely left-skewed tree. The worst-case time complexity satisfies the recurrence $W(n) = W(n-1)+O(n)$, which solves to $O(n^2)$. The best-case input is a sequence corresponding to a completely right-skewed tree, and the corresponding time complexity is $O(n)$. When the sequence corresponds to a balanced BST, the time complexity is given by $B(n) = 2B(n/2) + O(n)$, which solves to $O(n \log n)$.

The implementation above potentially iterates over nodes multiple times, which is wasteful. A better approach is to reconstruct the left subtree in the same iteration as identifying the nodes which lie in it. The code shown below takes this approach. Its worst-case time complexity is $O(n)$, since it performs a constant amount of work per node.

```
template <typename T>
shared_ptr<BinarySearchTree<T>> rebuild_BST_from_preorder_helper(
    const vector<T> &preorder, int &idx, const T &min, const T &max) {
  if (idx == preorder.size()) {
    return nullptr;
```

```
  }

  T curr = preorder[idx];
  if (curr < min || curr > max) {
    return nullptr;
  }

  ++idx;
  shared_ptr<BinarySearchTree<T> > root(new BinarySearchTree<T>{curr,
      rebuild_BST_from_preorder_helper(preorder, idx, min, curr),
      rebuild_BST_from_preorder_helper(preorder, idx, curr, max)});
  return root;
}

template <typename T>
shared_ptr<BinarySearchTree<T> > rebuild_BST_from_preorder(
    const vector<T> &preorder) {
  int idx = 0;
  return rebuild_BST_from_preorder_helper(preorder, idx,
                                          numeric_limits<T>::min(),
                                          numeric_limits<T>::max());
}
```

Problem 14.13, *pg. 109*: *Design an algorithm that takes a BST T of size n and height h, nodes s and b, and returns the LCA of s and b. Assume* s.key < b.key. *For example, in Figure 14.1 on Page 105, if s is node C and b is node G, your algorithm should return node B. Your algorithm should run in $O(h)$ time and $O(1)$ space. Nodes do not have pointers to their parents.*

Solution 14.13: In Solution 9.11 on Page 245 we presented an algorithm for this problem in the context of binary trees. The idea underlying that algorithm was to do a postorder walk—the LCA is the first node visited after s and b have both been visited.

This idea can be refined for BSTs—since nodes satisfy the BST property and keys are distinct, we prune much of the exploration. Specifically, initialize x to the root. If $x.key = s.key$, $x.key = b.key$, or $((s.key < x.key)$ and $(x.key < b.key))$ then the LCA is x itself. Otherwise, if $x.key > b.key$ we set x to $x.left$ and continue the search, since the LCA must lie in $x.left$. Similarly, if $x.key < s.key$ we set x to $x.right$ and continue.

```
template <typename T>
shared_ptr<BinarySearchTree<T>> find_LCA(
    shared_ptr<BinarySearchTree<T>> x,
    const shared_ptr<BinarySearchTree<T>> &s,
    const shared_ptr<BinarySearchTree<T>> &b) {
  while (x->data < s->data || x->data > b->data) {
    if (x->data < s->data) {
      x = x->right;  // LCA must be in x's right child
    }
    if (x->data > b->data) {
      x = x->left;  // LCA must be in x's left child
    }
```

```
  }

  // x->data >= s->data && x->data <= b->data
  return x;  // x is LCA
}
```

Problem 14.14, *pg. 109*: *Let r, s, and m be distinct nodes in a BST. In this BST, nodes do not have pointers to their parents and all keys are unique. Write a function which returns* `true` *if m has both an ancestor and a descendant in the set* $\{r, s\}$. *For example, in Figure 14.1 on Page 105, if m is Node J, your function should return* `true` *if the given set is* $\{A, K\}$ *and return* `false` *if the given set is* $\{I, P\}$.

Solution 14.14: There are two possibilities: m is a descendant of r and an ancestor of s, or m is an ancestor of r and a descendant of s.

Consider the first case. We can check if m is a descendant of r, and s is a descendant of m by simply doing one search for *s.key* in the subtree rooted at r, and recording whether m was encountered during the search. The search from r has time complexity $O(h)$, where h is the height of the tree, since we can use the BST property to prune one of the two children at each node. To check if m is an descendant of s, and r is a descendant of m, we do a symmetric search for *r.key* in the subtree rooted at s.

The disadvantage of performing these two searches one-after-another is that even when the distance between r and s is short, we may begin the search from the lower of the two, and incur the full $O(h)$ time complexity. We can prevent this by performing the searches from s for *r.key* and from r for *s.key* in an interleaved way; this way, if the final result returned is `true`, we will avoid performing an unsuccessful search on a large subtree.

```
template <typename T>
bool is_r_s_descendant_ancestor_of_m(
    const shared_ptr<BinarySearchTree<T>> &r,
    const shared_ptr<BinarySearchTree<T>> &s,
    const shared_ptr<BinarySearchTree<T>> &m) {
  shared_ptr<BinarySearchTree<T>> curr_r = r, curr_s = s;

  // Interleaving searches from r and s
  while (curr_r && curr_r != s && curr_s && curr_s != r) {
    if (curr_r == m || curr_s == m) {
      return true;
    }
    curr_r = curr_r->data > s->data ? curr_r->left : curr_r->right;
    curr_s = curr_s->data > r->data ? curr_s->left : curr_s->right;
  }

  // Keep searching from r
  while (curr_r && curr_r != s) {
    if (curr_r == m) {
      return true;
    }
    curr_r = curr_r->data > s->data ? curr_r->left : curr_r->right;
  }
```

```
  // Keep searching from s
  while (curr_s && curr_s != r) {
    if (curr_s == m) {
      return true;
    }
    curr_s = curr_s->data > r->data ? curr_s->left : curr_s->right;
  }
  return false;
}
```

Problem 14.15, *pg. 109*: *How would you efficiently perform a range query on a BST? Specifically, write a function that takes as input a BST and a range $[L, U]$ and returns a list of all the keys that lie in $[L, U]$?*

Solution 14.15: We can return the list of nodes whose entries lie in $[L, U]$ by first finding the first node l whose entry is greater than or equal to L. Node l can be found by applying the technique of Solution 14.4 on Page 311. If this returns `null` (because no node has an entry equal to L), l is the first node with a key greater than L; this can be computed via a slightly modified version of Solution 14.5 on Page 312. Note that l may be `null` (if all node entries are less than L), in which case we return the empty list.

If l is not `null`, we make it the start of the result list. Now repeatedly call the successor function (Solution 14.2 on Page 308), adding successive entries to the list, stopping when the successor function returns `null` (which is the case if all node entries are less than or equal to U), or a node whose entry is greater than U.

The time complexity to find l is $O(h)$, where h is the height of the tree. Individual calls to the successor function have complexity $O(h)$, which leads to a $O(hm)$ bound overall, where m is the size of the list.

However, the bound is not tight—m successive calls to successor have time complexity $O(h+m)$. The reason is that we traverse less than or equal to $d_l + 2(m-1) + d_u$ edges, where d_l and d_u are the depths of l and the last node whose key is less than or equal to U. Both d_l and d_u are bounded by h, leading to the claimed time complexity.

```
template <typename T>
shared_ptr<BinarySearchTree<T>> find_first_larger_equal_k(
    const shared_ptr<BinarySearchTree<T>> &r, const T &k) {
  if (!r) {
    return nullptr;
  } else if (r->data >= k) {
    // Recursively search the left subtree for first one >= k
    auto n = find_first_larger_equal_k(r->left, k);
    return n ? n : r;
  }
  // r->data < k so search the right subtree
  return find_first_larger_equal_k(r->right, k);
}

template <typename T>
list<shared_ptr<BinarySearchTree<T>>> range_query_on_BST(
```

```
    shared_ptr<BinarySearchTree<T>> n, const T &L, const T &U) {
  list<shared_ptr<BinarySearchTree<T>>> res;
  for (auto it = find_first_larger_equal_k(n, L);
       it && it->data <= U;
       it = find_successor_BST(it)) {
    res.emplace_back(it);
  }
  return res;
}
```

This solution can be improved by computing the number of entries in a BST in a range $[L, U]$ without enumerating all the entries in that range. See Solution 14.22 on Page 331 for details.

Problem 14.16, *pg. 110*: *Design an algorithm that takes three sorted arrays A, B, and C and returns a triple (i, j, k) such that distance(i, j, k) is minimum. Your algorithm should run in $O(|A| + |B| + |C|)$ time.*

Solution 14.16: We follow an approach similar to merge sort. Specifically, we keep three index variables, one for each of A, B, and C. These variables are initialized to 0, i.e., they index the minimum elements of A, B, and C. We iteratively identify the index variable whose corresponding element is the minimum of three, breaking ties by giving preference to A over B over C, and advance that corresponding index. In each iteration we record the difference between the largest and the smallest of the three elements and track the minimum difference m seen, along with the corresponding indices.

We claim that after all elements are processed, m and its associated triple are distance minimizing.

Proof:

Clearly m is always an upper bound on the minimum distance. Let (i, j, k) be a distance minimizing triple. Without loss of generality, assume (1.) $A[i] \leq B[j] \leq C[k]$, and (2.) $A[i] < A[i+1]$ and $C[k] > C[k-1]$. Since (i, j, k) is optimum, there cannot exist i' such that $A[i'] \in (A[i], C[k])$, or k' such that $C[k'] \in (A[i], C[k])$. (There may exist one or more $j' \neq j$ such that $B[j'] \in (A[i], C[k])$.)

Our algorithm will process $A[i]$ before $C[k]$, and by the observation in the previous paragraph, no other element of A or C will be processed after $A[i]$ is processed and before $C[k]$ is processed. Since $A[i] \leq B[j] \leq C[k]$, index $B[j]$ will be processed after $A[i]$ is processed and before $C[k]$ is processed. When $C[k]$ is processed, the index variable for B must correspond to an element of B which lies in $[A[i], C[k]]$. Therefore, when $C[k]$ is processed, the minimum difference will either already be $C[k] - A[i]$, or updated to $C[k] - A[i]$, i.e., the algorithm computes the correct result.

In the following code, we implement a general purpose function which finds the minimum distance in k sorted arrays. These arrays are passed in as `arrs`. Since we need to repeatedly find the minimum among all those sorted arrays, we use a balanced BST to identify the array that contains the minimum element. The BST

also allows us to find the difference of the minimum and maximum values in the collection efficiently. The overall time complexity is $O(n \log k)$, where n is the total number of elements in the k arrays. For the special case $k = 3$ specified in the problem statement, the time complexity is $O(n \log 3) = O(n)$.

```
template <typename T>
class ArrData {
  public:
    int idx;
    T val;

    const bool operator<(const ArrData &a) const {
      if (val != a.val) {
        return val < a.val;
      } else {
        return idx < a.idx;
      }
    }
};

template <typename T>
T find_min_distance_sorted_arrays(const vector<vector<T>> &arrs) {
  // Pointers for each of arrs
  vector<int> idx(arrs.size(), 0);
  T min_dis = numeric_limits<T>::max();
  set<ArrData<T>> current_heads;

  // Each of arrs puts its minimum element into current_heads
  for (int i = 0; i < arrs.size(); ++i) {
    if (idx[i] >= arrs[i].size()) {
      return min_dis;
    }
    current_heads.emplace(ArrData<T>{i, arrs[i][idx[i]]});
  }

  while (true) {
    min_dis = min(min_dis, current_heads.crbegin()->val -
                           current_heads.cbegin()->val);
    int tar = current_heads.cbegin()->idx;
    // Return if there is no remaining element in one array
    if (++idx[tar] >= arrs[tar].size()) {
      return min_dis;
    }
    current_heads.erase(current_heads.begin());
    current_heads.emplace(ArrData<T>{tar, arrs[tar][idx[tar]]});
  }
}
```

Problem 14.17, *pg. 110*: *You are to implement methods to analyze log file data to find the most visited pages. Specifically, implement the following methods:*

- `void add(Entry p)`—*add* `p.page` *to the set of visited pages. It is guaranteed that if* `add(q)` *is called after* `add(p)` *then* `q.timestamp` *is greater than or equal*

to `p.timestamp`.

– `List<String> common(k)`—*return a list of the* `k` *most common pages.*

First solve this problem when `common(k)` *is called exactly once after all pages have been read. Then solve the problem when calls to* `common` *and* `add` *are interleaved. Assume you have unlimited RAM.*

Solution 14.17: For the first scenario, we keep a hash table H of (*page, count*) pairs. We process the log file entry-by-entry, inserting a page into H with a count of 1 if the page is not already present; otherwise we increment the count for the page. After all pages have been read we can compute the k most common pages by iterating through the pairs in H and using, for example, the techniques in 11.13 on Page 270 or 11.14 on Page 271.

When calls to `add` and `common` are interleaved it is more efficient to use a hash table M in conjunction with a BST B. The BST stores objects which consist of a frequency and a page—compares are made based on the frequency field, with ties being broken by the page field. (The tie-breaker is needed to ensure two pages with the same frequency have distinct entries in B.) The hash table M maps each page to its corresponding object in B. To add a page p we first do a find in M. If p is present, the frequency field in the corresponding object in B is updated. (The simplest way to update B is to do a delete followed by an insert.) Otherwise a new object with frequency equal to 1 and page equal to p is added to B. The time complexity of `add` is dominated by the BST update, which is $O(\log n)$, where n is the number of distinct pages processed so far.

The `common` method is implemented by finding the maximum element in B and making $k-1$ calls to the predecessor function. If B is balanced, the time complexity of $k-1$ calls to predecessor is $O(k + \log n)$. For $k \ll n$ this compares very favorably with having to iterate through the entire collection as we did in the first scenario. The penalty is the added overhead of the BST B, specifically the increased time to perform each `add`.

Problem 14.18, *pg. 110*: *Implement the API in Problem 14.17 on Page 110. If* `common` *is called after processing the i-th entry,* `common` *should return the k most visited pages whose timestamp is in* $[t_i - W, t_i]$. *Here* t_i *is the timestamp of the i-th entry and W is specified by the client before any pages are read and does not change. RAM is limited—in particular you cannot keep a map containing all pages. Maximize time efficiency assuming calls to* `add` *and* `common` *may be interleaved and* `common` *is frequently called.*

Solution 14.18: Define the current window when entry i is being processed to be the time interval $[t_i - W, t_i]$.

We use three data structures:

1. A queue Q containing the entries whose timestamp is in the current window.
2. A BST B containing page-frequency pairs, where elements are ordered by frequency.
3. A hash table M mapping pages to entries in the BST B.

The k largest elements in B are the desired pages for the current interval.

The add function enqueues p in Q and increments the corresponding page's frequency in B. The timestamp on p may result in the current window changing. We examine the head of the queue Q, and iteratively remove entries whose timestamp is outside the current window. For each entry that is removed, we reduce the frequency of the corresponding page in B. To do this, we use the hash table M to go from p's *visited_page* field to the corresponding entry in B.

The common function is implemented exactly as in Solution 14.17 on the preceding page.

The time complexity for adding pages is dominated by updates to the BST. In the worst case, every page is unique and all appear in the window, leading to a $O(n \log n)$ time complexity, where n is the number of log entries. The space complexity is $O(n)$.

In practical settings, the maximum number of pages in a window, and hence in B, will likely be much less than n. If the number of entries in a window is bounded by c, then the time complexity for n calls to add is $O(n \log c)$—the $\log c$ term corresponds to the time needed to perform BST updates. The space complexity is $O(c)$.

Problem 14.19, *pg. 111*: *Write a function that takes a single integer argument n and computes all the Gaussian integers $a + bi$, for $-n \leq a, b \leq n$ that are Gaussian primes.*

Solution 14.19: The *modulus* of a complex number $z = a + bi$ is by definition $\sqrt{a^2 + b^2}$, and is commonly denoted by $|z|$.

It is straightforward to see that $|z_1 z_2| = |z_1||z_2|$. Therefore, one approach to computing the Gaussian primes in the desired range is to sort the numbers in the range based on their modulus. The only numbers with modulus 1 are the units. The next smallest modulus is $\sqrt{2}$, e.g., $1 + i$. All numbers whose modulus is $\sqrt{2}$ must be Gaussian primes; this follows from the primality of 2 in the conventional integers. We can eliminate all multiples of such numbers by nonunits, and then examine the remaining candidates for the one with the smallest modulus, and continue. We maintain the set of candidates in a BST using the modulus to order numbers, breaking ties lexicographically. This approach is the analog of the sieve method for ordinary primes, described in Solution 5.11, extended to two dimensions.

```
bool is_unit(const complex<int> &z) {
  return (z.real() == 1 && z.imag() == 0) ||
         (z.real() == -1 && z.imag() == 0) ||
         (z.real() == 0 && z.imag() == 1) ||
         (z.real() == 0 && z.imag() == -1);
}

class ComplexCompare {
  public:
    const bool operator()(const complex<double> &lhs,
                          const complex<double> &rhs) const {
      if (norm(lhs) != norm(rhs)) {
        return norm(lhs) < norm(rhs);
      } else if (lhs.real() != rhs.real()) {
        return lhs.real() < rhs.real();
      } else {
        return lhs.imag() < rhs.imag();
```

```
      }
    }
};

vector<complex<int>> generate_Gaussian_primes(const int &n) {
  set<complex<double>, ComplexCompare> candidates;
  vector<complex<int>> primes;

  // Generate all possible Gaussian prime candidates
  for (int i = -n; i <= n; ++i) {
    for (int j = -n; j <= n; ++j) {
      if (is_unit({i, j}) == false && abs(complex<double>(i, j)) != 0) {
        candidates.emplace(i, j);
      }
    }
  }

  while (candidates.empty() == false) {
    complex<double> p = *(candidates.begin());
    candidates.erase(candidates.begin());
    primes.emplace_back(p);
    int max_multiplier = n / floor(sqrt(norm(p))) + 1;

    for (int i = max_multiplier; i >= -max_multiplier; --i) {
      for (int j = max_multiplier; j >= -max_multiplier; --j) {
        complex<double> x = {i, j};
        if (is_unit(x) == false) {
          candidates.erase(x * p);
        }
      }
    }
  }
  return primes;
}
```

Problem 14.20, *pg. 111*: *Implement a function that computes the view from above. Your input is a sequence of line segments, each specified as a 4-tuple $\langle l, r, c, h \rangle$, where l and r are the left and right endpoints, respectively, c encodes the color, and h are the height. The output should be in the same format. No two segments whose intervals overlap have the same height.*

Solution 14.20: First observe that the left endpoint and the right endpoint of each segment in the view from above is the left or right endpoint of an input segment.

This observation leads to the following algorithm. Sort the endpoints of the segments and then do a sweep from left-to-right. As we sweep, we maintain the set of segments that intersect the current position; this set is stored in a BST with the height being the key. The color is determined by the highest segment. When we encounter a left endpoint, we add the corresponding segment in a BST. When we encounter a right endpoint, we remove the corresponding segment from the BST. We use the height field as a proxy for the segment, since the problem statement guarantees that the height uniquely determines the segment.

```
template <typename XaxisType, typename ColorType, typename HeightType>
class LineSegment {
  public:
    XaxisType left, right;  // specifies the interval
    ColorType color;
    HeightType height;

    const bool operator<(const LineSegment &that) const {
      return height < that.height;
    }
};

template <typename XaxisType, typename ColorType, typename HeightType>
class Endpoint {
  public:
    bool isLeft;
    const LineSegment<XaxisType, ColorType, HeightType>* l;

    const bool operator<(const Endpoint &that) const {
      return val() < that.val();
    }

    const XaxisType &val(void) const {
      return isLeft ? l->left : l->right;
    }
};

template <typename XaxisType, typename ColorType, typename HeightType>
void calculate_view_from_above(
    const vector<LineSegment<XaxisType, ColorType, HeightType>> &A) {
  vector<Endpoint<XaxisType, ColorType, HeightType>> E;
  for (int i = 0; i < A.size(); ++i) {
    E.emplace_back(Endpoint<XaxisType, ColorType, HeightType>{true, &A[i]});
    E.emplace_back(Endpoint<XaxisType, ColorType, HeightType>{false, &A[i]});
  }
  sort(E.begin(), E.end());

  XaxisType prev_xaxis = E.front().val();  // the first left end point
  shared_ptr<LineSegment<XaxisType, ColorType, HeightType>> prev = nullptr;
  map<HeightType, const LineSegment<XaxisType, ColorType, HeightType>*> T;
  for (const Endpoint<XaxisType, ColorType, HeightType> &e: E) {
    if (T.empty() == false && prev_xaxis != e.val()) {
      if (prev == nullptr) {  // found first segment
        prev = shared_ptr<LineSegment<XaxisType, ColorType, HeightType>>(
          new LineSegment<XaxisType, ColorType, HeightType>{
            prev_xaxis, e.val(), T.crbegin()->second->color,
            T.crbegin()->second->height});
      } else {
        if (prev->height == T.crbegin()->second->height &&
            prev->color == T.crbegin()->second->color) {
          prev->right = e.val();
        } else {
          cout << "[" << prev->left << ", " << prev->right << "]"
               << ", color = " << prev->color << ", height = "
```

```
                << prev->height << endl;
            *prev = {prev_xaxis, e.val(), T.crbegin()->second->color,
                     T.crbegin()->second->height};
          }
        }
      }
      prev_xaxis = e.val();

      if (e.isLeft == true) {  // left end point
        T.emplace(e.l->height, e.l);
      } else {  // right end point
        T.erase(e.l->height);
      }
    }

    // Output the remaining segment if any
    if (prev) {
      cout << "[" << prev->left << ", " << prev->right << "]"
           << ", color = " << prev->color << ", height = "
           << prev->height << endl;
    }
  }
```

ϵ-Variant 14.20.1: Solve the same problem when multiple segments may have the same height. Break ties arbitrarily.

ϵ-Variant 14.20.2: Design an efficient algorithm for computing the length of the union of a set of closed intervals.

Variant 14.20.3: Design an efficient algorithm for computing the area of a set of rectangles whose sides are aligned with the X and Y axes.

Variant 14.20.4: Runners $R_1, R_2, \ldots, R_n$ race on a track of length L. Runner R_i begins at an offset s_i from the start of the track, and runs at speed v_i. Compute the set of runners who lead the race at some time.

Variant 14.20.5: Given a set H of nonintersecting horizontal line segments in the 2D plane, and a set V of nonintersecting vertical line segments in the 2D plane, determine if any pair of line segments intersect.

Problem 14.21, *pg. 112*: *Design a data structure that implements the following methods:*

- `insert(s,c)`, *which adds client s with credit c, overwriting any existing entry for s.*
- `remove(s)`, *which removes client s.*
- `lookup(s)`, *which returns the number of credits associated with client s, or* -1 *if s is not present.*
- `addAll(C)`, *the effect of which is to increment the number of credits for each client currently present by C.*

– `max()`, *which returns the client with the highest number of credits.*

The `insert(s,c)`, `remove(s)`, *and* `lookup(s)` *methods should run in time* $O(\log n)$, *where* n *is the number of clients. The remaining methods should run in time* $O(1)$.

Solution 14.21: We use one hash table, `credits` and one BST, `inverse_credits`. We also use an integer-valued variable `offset` which is initialized to 0. A call to `addAll(C)` increments `offset` by `C`.

The hash table `credits` consists of key-value pairs, where the key is the client string and value is an integer. It gives us the ability to do lookup in $O(1)$ time. The `lookup(s)` method returns the sum of `offset` and the value associated with `s` in `credits`. For `lookup(s)` to work correctly, we must subtract `offset` from `c` when performing `insert(s, c)`—this way a client's credits include only the credits added via `addAll` after it was inserted.

The BST `inverse_credits` is used to implement `max()` in $O(1)$ time. The entries in `inverse_credits` are key-value pairs, where each key is a value v from `credits`; its associated value is a hash table of the client strings which v is associated with in `credits`. The `insert(s, c)` and `remove(s)` methods entail updating `inverse_credits`. Insertion consists of a lookup in `inverse_credits` followed by either adding a new pair to `inverse_credits` (if `c - offset` is not associated with any key in `credits`) or adding a string to the hash table associated with `c - offset`. The time complexity for updating `inverse_credits` when `insert(s, c)` is called is $O(\log n)$ for the BST lookup plus $O(1)$ for the hash table creation or update, i.e., $O(\log n)$. Similarly, the time complexity for updating `inverse_credits` when `remove(s)` is called is $O(\log n)$. The detailed implementation is given below. We take advantage of STL's implementation of BST, in which the minimum and maximum entries are computed in constant time, to make `max()` an $O(1)$ time operation.

```
class ClientsCreditsInfo {
  private:
    int offset;
    unordered_map<string, int> credits;
    map<int, unordered_set<string>> inverse_credits;

  public:
    ClientsCreditsInfo(void) : offset(0) {}

    void insert(const string &s, const int &c) {
      credits.emplace(s, c - offset);
      inverse_credits[c - offset].emplace(s);
    }

    void remove(const string &s) {
      auto credits_it = credits.find(s);
      if (credits_it != credits.end()) {
        inverse_credits[credits_it->second].erase(s);
        credits.erase(credits_it);
      }
    }

    int lookup(const string &s) const {
```

```
    auto it = credits.find(s);
    return it == credits.cend() ? -1 : it->second + offset;
  }

  void addAll(const int &C) {
    offset += C;
  }

  string max(void) const {
    auto it = inverse_credits.crbegin();
    return it == inverse_credits.crend() || it->second.empty() ?
           "" : *it->second.cbegin();
  }
};
```

Problem 14.22, *pg. 112*: *Suppose each node in a BST has a size field, which denotes the number of nodes at the subtree rooted at that node, inclusive of the node. How would you efficiently compute the number of nodes that lie in a given range? Can the size field be updated efficiently on insert and on delete?*

Solution 14.22: We can find the number of nodes whose entries lie in $[L, U]$ by using the approach of Solution 14.15 on Page 322, and simply counting nodes, rather than inserting them into a list. This leads to the same time complexity. However, we can do better by exploiting the size field.

For example, suppose we want to find the number of entries that are less than a given value v. Initialize the count to 0. We search for the first occurrence of v, and each time we take a left child, we leave count unchanged; each time we take a right child, we add one plus the size of the corresponding left child. If v is present, when we reach the first occurrence of v, we add the size of v's left child. The same approach can be used to find the number of entries that are greater than v.

The time bound for these computations is $O(h)$, since the search always descends the tree. We can compute the number of nodes in the final result by first computing the number of nodes less than L and the number of nodes greater than H, and subtracting that from the total number of nodes (which is the size stored at the root).

The size field can be updated on insert and delete without changing the $O(h)$ time complexity of both. Essentially, the only nodes whose size field change are those on the search path to the added/deleted node. Some conditional checks are needed for each such node, but these add constant time per node, leaving the $O(h)$ time complexity unchanged.

Variant 14.22.1: Define the "Markowitz bullet" of a set of points P in the upper right quadrant of the Cartesian plane to be those points which are not below and to the right of any other point in P. Design a data structure for representing the Markowitz bullet. Specifically, it should be possible to efficiently check if a new point is below and to the right of some point in the Markowitz bullet, and to add a point to the Markowitz bullet (which may result in other points being removed from the bullet).

Problem 14.23, *pg. 113*: *Design a data structure that stores closed intervals and can efficiently return the complete set of intervals that intersect a specified range $[L, U]$. Your data structure must also support efficient insertions and deletions.*

Solution 14.23: The solution to this problem is based on the notion of an interval tree. This is a BST in which entries are intervals, e.g., $[213, 455]$. For simplicity we assume closed intervals throughout. Given an interval $I = [L, U]$, we will refer to L as the left endpoint of I, and U as the right endpoint of I. We use the left endpoint of the interval as the BST key.

Each node a stores an interval $[l_a, u_a]$, and also a max field m_a, which is the largest right endpoint amongst the intervals stored in the subtree rooted at a. It is fairly straightforward to show that the max field can be updated through inserts and deletes without changing the time complexity of these operations.

Searching for a node that intersects $I = [L, U]$ is done as follows. If the root r is `null` or $[l_r, u_r]$ has a nonempty intersection with I, return r; otherwise, if r's left child *r.left* is not `null` and the max field of *r.left* is greater than or equal to L, recurse on *r.left*; otherwise recurse on r's right child.

Since it descends the tree at each step, and performs constant work within a step, the procedure has time complexity $O(h)$ where h is the height of the tree. Its correctness follows from basic facts about intervals. The only tricky case is the justification that we do not need to search r's right subtree when the max field M of *r.left* is greater than or equal to L. The reasoning is as follows. Suppose no interval in *r.left* overlaps with I. We know there must be at least one interval of the form $[m, M]$ in *r.left*. Since I does not intersect any interval in *r.left*, it must be that $U < m$ or $L > M$. The latter is not possible, because the max field M is greater or equal to than L. Since the tree satisfies the BST property on the left end points, we know that for each $[p, q]$ in r's right subtree, $U < p$, and hence cannot intersect any interval in that subtree.

The problem statement asked for all intervals in the tree that intersect the given interval. We can do this by iteratively finding and removing the node returned in the procedure above, and later putting the nodes back in the tree. The time complexity is $O(mh)$, where m is the number of nodes in the result.

Variant 14.23.1: Solve the same problem without modifying the tree.

Problem 15.1, *pg. 115*: *Design an efficient algorithm for computing the skyline.*

Solution 15.1: The simplest solution is to compute the skyline incrementally. For one building, the skyline is trivial. Suppose we know the skyline for $n-1$ buildings, and need to compute the new skyline when the n-th building (L_n, R_n, H_n) is added. We now iterate through the existing skyline from left to right to see where L_n should be added. Next we move through the existing skyline and increase any heights that are less than H_n to H_n until we reach R_n.

This algorithm is simple, but has $\Theta(n^2)$ complexity, since adding the n-th building may entail $\Theta(n)$ comparisons. A better solution is to use divide and conquer: we compute skylines for the first $\frac{n}{2}$ buildings and the last $\frac{n}{2}$ buildings, and merge the

results. The merge is similar to the procedure for adding a single building, described above, and can be performed in $O(n)$ time. Basically, we iterate through the two skylines together from left-to-right, matching their left and right coordinates, and adjusting heights appropriately.

```
template <typename CoordType, typename HeightType>
class Skyline {
  public:
    CoordType left, right;
    HeightType height;
};

template <typename CoordType, typename HeightType>
void merge_intersect_skylines(vector<Skyline<CoordType, HeightType>> &merged,
                              Skyline<CoordType, HeightType> &a, int &a_idx,
                              Skyline<CoordType, HeightType> &b, int &b_idx) {
  if (a.right <= b.right) {
    if (a.height > b.height) {
      if (b.right != a.right) {
        merged.emplace_back(a), ++a_idx;
        b.left = a.right;
      } else {
        ++b_idx;
      }
    } else if (a.height == b.height) {
      b.left = a.left, ++a_idx;
    } else {  // a.height < b.height
      if (a.left != b.left) {
        merged.emplace_back(
          Skyline<CoordType, HeightType>{a.left, b.left, a.height});
      }
      ++a_idx;
    }
  } else {  // a.right > b.right
    if (a.height >= b.height) {
      ++b_idx;
    } else {
      if (a.left != b.left) {
        merged.emplace_back(
          Skyline<CoordType, HeightType>{a.left, b.left, a.height});
      }
      a.left = b.right;
      merged.emplace_back(b), ++b_idx;
    }
  }
}

template <typename CoordType, typename HeightType>
vector<Skyline<CoordType, HeightType>> merge_skylines(
    vector<Skyline<CoordType, HeightType>> &L,
    vector<Skyline<CoordType, HeightType>> &R) {
  int i = 0, j = 0;
  vector<Skyline<CoordType, HeightType>> merged;

```

```
  while (i < L.size() && j < R.size()) {
    if (L[i].right < R[j].left) {
      merged.emplace_back(L[i++]);
    } else if (R[j].right < L[i].left) {
      merged.emplace_back(R[j++]);
    } else if (L[i].left <= R[j].left) {
      merge_intersect_skylines(merged, L[i], i, R[j], j);
    } else {  // L[i].left > R[j].left
      merge_intersect_skylines(merged, R[j], j, L[i], i);
    }
  }

  copy(L.cbegin() + i, L.cend(), back_inserter(merged));
  copy(R.cbegin() + j, R.cend(), back_inserter(merged));
  return merged;
}

template <typename CoordType, typename HeightType>
vector<Skyline<CoordType, HeightType>> drawing_skylines_helper(
    vector<Skyline<CoordType, HeightType>> &skylines,
    const int &start, const int &end) {
  if (end - start <= 1) {  // 0 or 1 skyline, just copy it
    return {skylines.cbegin() + start, skylines.cbegin() + end};
  }
  int mid = start + ((end - start) >> 1);
  auto L = drawing_skylines_helper(skylines, start, mid);
  auto R = drawing_skylines_helper(skylines, mid, end);
  return merge_skylines(L, R);
}

template <typename CoordType, typename HeightType>
vector<Skyline<CoordType, HeightType>> drawing_skylines(
    vector<Skyline<CoordType, HeightType>> skylines) {
  return drawing_skylines_helper(skylines, 0, skylines.size());
}
```

Variant 15.1.1: Compute the skyline problem when each building has the shape of an isosceles triangle with a 90 degree angle at its apex.

Problem 15.2, *pg. 115*: *Design an efficient algorithm that takes an array A of n numbers and returns the number of inverted pairs of indices.*

Solution 15.2: The brute-force algorithm examines all $i \in [0, n-1]$ and all $j \in [i+1, n-1]$, and has an $O(n^2)$ complexity. A more efficient approach is to use merge sort. Suppose we have counted the number of inversions in the left half L and the right half R of A. What are the inversions that remain to be counted? Sort the left and right half arrays, and merge the two halves. For any (i, j) pair, if $L[i] > R[j]$, then for all $i' \geq i$ we must have $L[i'] > R[j]$, and we need to add $m - i$ to the inversion count, where m is the length of L. The time complexity is identical to that for merge sort, i.e., $O(n \log n)$.

```
template <typename T>
int merge(vector<T> &A, const int &start, const int &mid, const int &end) {
  vector<T> sorted_A;
  int left_start = start, right_start = mid, inver_count = 0;

  while (left_start < mid && right_start < end) {
    if (A[left_start] <= A[right_start]) {
      sorted_A.emplace_back(A[left_start++]);
    } else {
      // A[left_start:mid - 1] will be the inversions
      inver_count += mid - left_start;
      sorted_A.emplace_back(A[right_start++]);
    }
  }
  copy(A.begin() + left_start, A.begin() + mid, back_inserter(sorted_A));
  copy(A.begin() + right_start, A.begin() + end, back_inserter(sorted_A));

  // Update A with sorted_A
  copy(sorted_A.begin(), sorted_A.end(), A.begin() + start);
  return inver_count;
}

template <typename T>
int count_inversions_helper(vector<T> &A, const int &start, const int &end) {
  if (end - start <= 1) {
    return 0;
  }

  int mid = start + ((end - start) >> 1);
  return count_inversions_helper(A, start, mid) +
         count_inversions_helper(A, mid, end) + merge(A, start, mid, end);
}

template <typename T>
int count_inversions(vector<T> A) {
  return count_inversions_helper(A, 0, A.size());
}
```

Problem 15.3, *pg. 116*: *You are given a list of pairs of points in the two-dimensional Cartesian plane. Each point has integer x and y coordinates. How would you find the two closest points?*

Solution 15.3: The brute-force solution is to consider all pairs of points: this yields an $O(n^2)$ algorithm.

To improve upon the brute-force solution, it is instructive to consider the one-dimensional case. The obvious solution for the one-dimensional case is to iterate through the points in sorted order, comparing the distance between successive points with the running minimum. However, this does not generalize to the two-dimensional case, since there is no natural total ordering of the points. Another approach for the one-dimensional case is divide and conquer: partition the set about the median, solve the problem for the left and right partitions, and combine the

results. The last step entails finding points closest to the median from the left and right partitions.

The complexity of the partitioning approach is the same as that of the approach based on sorting—$O(n \log n)$, where n is the number of points. However the partitioning approach is applicable in more than one dimension. Specifically, we can split the points into two equal-sized sets using a line $x = P$ parallel to the y-axis. (Such a line can be found by computing the median of the values for the x coordinates—this calculation can be performed using the algorithm described in Solution 11.13 on Page 270.)

We can then compute the closest pair of points recursively on the two sets; let the closest pair of points on the left of P be d_l apart and the closest pair of points to the right of P be d_r apart. Let $d = \min(d_l, d_r)$.

Now, all we need to look at is points which are in the band $[P - d, P + d]$. In degenerate situations, all points may be within this band. If we compare all the pairs, the complexity becomes quadratic again. However we can sort the points in the band on their y coordinates and iterate through the sorted list, looking for points d or less distance from the point being processed.

Intuitively, there cannot be many such points since otherwise, the closest pair in the left and right partitions would have to be less than d apart. This intuition can be analytically justified—Shamos and Hoey's famous 1975 paper *"Closest-point problems"* shows that no more than six points can be within d distance of any point which leads to an $O(n \log n)$ algorithm. (The time is dominated by the need to sort.)

The recursion can be sped up by switching to brute-force when a small number of points remain.

```
class Point {
  public:
    int x, y;
};

double distance(const Point &a, const Point &b) {
  return sqrt((a.x - b.x) * (a.x - b.x) + (a.y - b.y) * (a.y - b.y));
}

// Return the closest two points and its distance as a tuple
tuple<Point, Point, double> brute_force(const vector<Point> &P, const int &s,
                                        const int &e) {
  tuple<Point, Point, double> ret;
  get<2>(ret) = numeric_limits<double>::max();
  for (int i = s; i < e; ++i) {
    for (int j = i + 1; j < e; ++j) {
      double dis = distance(P[i], P[j]);
      if (dis < get<2>(ret)) {
        ret = {P[i], P[j], dis};
      }
    }
  }
  return ret;
}
```

```

// Return the closest two points and its distance as a tuple
tuple<Point, Point, double> find_closest_pair_in_remain(vector<Point> &P,
                                                        const double &d) {
  sort(P.begin(), P.end(), [](const Point &a, const Point &b) -> bool {
                              return a.y < b.y;
                            });

  // At most six points in P
  tuple<Point, Point, double> ret;
  get<2>(ret) = numeric_limits<double>::max();
  for (int i = 0; i < P.size(); ++i) {
    for (int j = i + 1; j < P.size() && P[j].y - P[i].y < d; ++j) {
      double dis = distance(P[i], P[j]);
      if (dis < get<2>(ret)) {
        ret = {P[i], P[j], dis};
      }
    }
  }
  return ret;
}

// Return the closest two points and its distance as a tuple
tuple<Point, Point, double> find_closest_pair_points_helper(
    const vector<Point> &P, const int &s, const int &e) {
  if (e - s <= 3) {  // brute-force to find answer if there are <= 3 points
    return brute_force(P, s, e);
  }

  int mid = (e + s) >> 1;
  auto l_ret = find_closest_pair_points_helper(P, s, mid);
  auto r_ret = find_closest_pair_points_helper(P, mid, e);
  auto min_l_r = get<2>(l_ret) < get<2>(r_ret) ? l_ret : r_ret;
  vector<Point> remain;  // stores the points whose x-dis < min_d
  for (const Point &p : P) {
    if (abs(p.x - P[mid].x) < get<2>(min_l_r)) {
      remain.emplace_back(p);
    }
  }

  auto mid_ret = find_closest_pair_in_remain(remain, get<2>(min_l_r));
  return get<2>(mid_ret) < get<2>(min_l_r) ? mid_ret : min_l_r;
}

pair<Point, Point> find_closest_pair_points(vector<Point> P) {
  sort(P.begin(), P.end(), [](const Point &a, const Point &b) -> bool {
                              return a.x < b.x;
                            });
  auto ret = find_closest_pair_points_helper(P, 0, P.size());
  return {get<0>(ret), get<1>(ret)};
}
```

Problem 15.4, *pg. 116*: *Design an efficient algorithm to compute the diameter of a tree.*

Solution 15.4: We can compute the diameter by running BFS, described on Page 132, from each node and recording the maximum value of the shortest path distances computed. This has $O(|V|(|V| + |E|)) = O(|V|^2)$ time complexity since $|E| = |V| - 1$ in a tree.

We can achieve better time complexity by using divide and conquer. First we define some notation. If T is a nonempty tree, let root(T) denote the node at the root of T. Let $l_{u,v}$ be the length of the edge (u, v). The *degree* of a node u in a rooted tree is the number of its children. Define the *weighted height* h_u of a tree rooted at u to be 0 if u is a leaf and $\max_{1 \le i \le n}(l_{u,\text{root}(T_i)} + h_{\text{root}(T_i)})$, where $T_1, T_2, \ldots, T_n$ are the subtrees rooted at u's children.

Let T be a tree whose root is r. Suppose r has degree m. For now, assume $m \ge 2$. Let $d_1, d_2, \ldots, d_m$ be the diameters and $h_1, h_2, \ldots, h_n$ the weighted heights of the subtrees.

Let λ be a longest path in T. Either it passes through r or it does not. If λ does not pass through r, it must be entirely within one of the m subtrees and hence the longest path length in T is the maximum of $d_1, d_2, \ldots, d_m$. If it does pass through r, it must be between a pair of nodes in distinct subtrees that are farthest from r. The distance from r to the node in T_i that is farthest from it is simply $f_i = h_i + l_{r,i}$, where $l_{u,v}$ denotes the length of the edge (u, v). Therefore the longest length path in T is the larger of the maximum of $d_1, d_2, \ldots, d_m$ and the sum of the two largest f_is.

Now we consider the cases $m = 0$ and $m = 1$. If $m = 0$ the subtree rooted at t is just the node t and the length of the longest path is 0. If $m = 1$ the length of the longest path in t is $\max(h_1 + l_{r,1}, d_1)$.

The following algorithm computes the tree diameter. Process the tree in bottom-up fashion. For each node we process its subtrees one at a time. We update the maximum tree diameter based on the subtree weighted heights, diameters, and edge weights, using the observations above. The time complexity is proportional to the size of the tree, i.e., $O(|V|)$.

```
class TreeNode {
  public:
    vector<pair<shared_ptr<TreeNode>, double>> edges;
};

// Return (height, diameter) pair
pair<double, double> compute_height_and_diameter(
    const shared_ptr<TreeNode> &r) {
  double diameter = numeric_limits<double>::min();
  array<double, 2> height = {0.0, 0.0};  // store the max 2 heights
  for (const pair<shared_ptr<TreeNode>, double> &e : r->edges) {
    pair<double, double> h_d = compute_height_and_diameter(e.first);
    if (h_d.first + e.second > height[0]) {
      height[1] = height[0];
      height[0] = h_d.first + e.second;
    } else if (h_d.first + e.second > height[1]) {
      height[1] = h_d.first + e.second;
    }
    diameter = max(diameter, h_d.second);
  }
  return {height[0], max(diameter, height[0] + height[1])};
```

```
}

double compute_diameter(const shared_ptr<TreeNode> &T) {
  return T ? compute_height_and_diameter(T).second : 0.0;
}
```

Problem 15.5, *pg. 118*: *Given a circular array A, compute its maximum subarray sum in $O(n)$ time, where n is the length of A. Can you devise an algorithm that takes $O(n)$ time and $O(1)$ space?*

Solution 15.5: First recall the standard algorithm for the conventional maximum subarray sum problem. This proceeds by computing the maximum subarray sum $S[i]$ when the subarray ends at i, which is $\max(S[i-1] + A[i], A[i])$. Its running time is $O(n)$, where n is the length of the array.

One approach for the maximum circular subarray is to break the problem into two separate instances. The first instance is the noncircular one, and is solved as described above.

The second instance entails looking for the maximum subarray that cycles around. Naïvely, this entails finding the maximum subarray that starts at index 0, the maximum subarray ending at index $n-1$, and adding their sums. However, these two subarrays may overlap, and simply subtracting out the overlap does not always give the right result (consider the array $\langle 10, -4, 5, -4, 10 \rangle$).

Instead, we compute for each i the maximum subarray sum S_i for the subarray that starts at 0 and ends at or before i, and the maximum subarray E_i for the subarray that starts after i and ends at the last element. Then the maximum subarray sum for a subarray that cycles around is the maximum over all i of $S_i + E_i$.

```
// Calculate the non-circular solution
template <typename T>
T find_max_subarray(const vector<T> &A) {
  T maximum_till = 0, maximum = 0;
  for (const T &a : A) {
    maximum_till = max(a, a + maximum_till);
    maximum = max(maximum, maximum_till);
  }
  return maximum;
}

// Calculate the solution which is circular
template <typename T>
T find_circular_max_subarray(const vector<T> &A) {
  // Maximum subarray sum starts at index 0 and ends at or before index i
  vector<T> maximum_begin;
  T sum = A.front();
  maximum_begin.emplace_back(sum);
  for (int i = 1; i < A.size(); ++i) {
    sum += A[i];
    maximum_begin.emplace_back(max(maximum_begin.back(), sum));
  }

```

```
  // Maximum subarray sum starts at index i + 1 and ends at the last element
  vector<T> maximum_end(A.size());
  maximum_end.back() = 0;
  sum = 0;
  for (int i = A.size() - 2; i >= 0; --i) {
    sum += A[i + 1];
    maximum_end[i] = max(maximum_end[i + 1], sum);
  }

  // Calculate the maximum subarray which is circular
  T circular_max = 0;
  for (int i = 0; i < A.size(); ++i) {
    circular_max = max(circular_max, maximum_begin[i] + maximum_end[i]);
  }
  return circular_max;
}

template <typename T>
T max_subarray_sum_in_circular(const vector<T> &A) {
  return max(find_max_subarray(A), find_circular_max_subarray(A));
}
```

Alternately, the maximum subarray that cycles around can be determined by computing the minimum subarray—the remaining elements yield a subarray that cycles around. (One or both of the first and last elements may not be included in this subarray, but that is fine.) This approach uses $O(1)$ space and $O(n)$ time; code for it is given below.

```
template <typename T>
T find_optimum_subarray_using_comp(const vector<T> &A,
                                   const T&(*comp)(const T&, const T&)) {
  T till = 0, overall = 0;
  for (const T &a : A) {
    till = comp(a, a + till);
    overall = comp(overall, till);
  }
  return overall;
}

template <typename T>
T max_subarray_sum_in_circular(const vector<T> &A) {
  // Find the max in non-circular case and circular case
  return max(find_optimum_subarray_using_comp(A, max),  // non-circular case
             accumulate(A.cbegin(), A.cend(), 0) -
             find_optimum_subarray_using_comp(A, min));  // circular case
}
```

Problem 15.6, *pg. 119*: *Given an array A of n numbers, find a longest subsequence $\langle i_0, \ldots, i_{k-1} \rangle$ such that $i_j < i_{j+1}$ and $A[i_j] \leq A[i_{j+1}]$ for any $j \in [0, k-2]$.*

Solution 15.6: We present two solutions, an $O(n^2)$, and an $O(n \log n)$ one.

We first describe the $O(n^2)$ solution. Let s_i be the length of the longest nondecreasing subsequence of A that ends at $A[i]$ (specifically, $A[i]$ is included in this

subsequence). Then we can write the following recurrence:

$$s_i = \max\left(\max_{j\in[0,i-1]}\left(\begin{array}{ll} s_j+1, & \text{if } A[j] \le A[i]; \\ 1, & \text{otherwise.} \end{array}\right), 1\right)$$

We use this recurrence to fill up a table for s_i. The time complexity of this algorithm is $O(n^2)$. If we want the sequence as well, for each i, in addition to storing the length of the sequence, we store the index of the last element of sequence that we extended to get the current sequence. Here is an implementation of this algorithm:

```
template <typename T>
vector<T> longest_nondecreasing_subsequence(const vector<T> &A) {
  // Empty array
  if (A.empty() == true) {
    return A;
  }

  vector<int> longest_length(A.size(), 1), previous_index(A.size(), -1);
  int max_length_idx = 0;
  for (int i = 1; i < A.size(); ++i) {
    for (int j = 0; j < i; ++j) {
      if (A[i] >= A[j] && longest_length[j] + 1 > longest_length[i]) {
        longest_length[i] = longest_length[j] + 1;
        previous_index[i] = j;
      }
    }
    // Record the index where longest subsequence ends
    if (longest_length[i] > longest_length[max_length_idx]) {
      max_length_idx = i;
    }
  }

  // Build the longest nondecreasing subsequence
  int max_length = longest_length[max_length_idx];
  vector<T> ret(max_length);
  while (max_length > 0) {
    ret[--max_length] = A[max_length_idx];
    max_length_idx = previous_index[max_length_idx];
  }
  return ret;
}
```

We now describe a subtler algorithm that has $O(n \log n)$ complexity. Let $M_{i,j}$ be the smallest possible tail value for any nondecreasing subsequence of length j using array elements $A[0], A[1], \ldots, A[i]$. Note that for any i, we must have $M_{i,1} \le M_{i,2} \le \cdots \le M_{i,j}$.

We process A's elements iteratively. When processing $A[i+1]$, we look for the largest j such that $M_{i,j} \le A[i+1]$. First, assume such a j exists. Then we can construct a $j+1$ length subsequence that ends at $A[i+1]$. If no length $j+1$ nondecreasing subsequence exists in $A[0], A[1], \ldots, A[i]$, then $M_{i+1,j+1}$ must be $A[i+1]$, otherwise it remains equal to $M_{i,j+1}$. Furthermore, $M_{i+1,j'}$ remains unchanged for all $j' \le j$.

Now suppose there does not exist j such that $M_{i,j} \le A[i+1]$. This can only be true

if $A[i+1]$ is the unique smallest element in $A[0:i+1]$. Therefore we set $M_{i+1,1}$ to $A[i+1]$.

Therefore processing $A[i+1]$ entails a binary search for j and then an update to $M_{i+1,j+1}$ if possible, leading to an $O(n \log n)$ time complexity.

Code implementing this procedure is given below; the appropriate entries from M are maintained in the `tail_values` vector.

```
template <typename T>
int longest_nondecreasing_subsequence(const vector<T> &A) {
  vector<T> tail_values;
  for (const T &a : A) {
    auto it = upper_bound(tail_values.begin(), tail_values.end(), a);
    if (it == tail_values.end()) {
      tail_values.emplace_back(a);
    } else {
      *it = a;
    }
  }
  return tail_values.size();
}
```

ϵ-Variant 15.6.1: Define a sequence of numbers $\langle a_0, a_1, \ldots, a_{n-1} \rangle$ to be *alternating* if $a_i < a_{i+1}$ for even i and $a_i > a_{i+1}$ for odd i. Given an array of numbers A of length n, find a longest subsequence $\langle i_0, \ldots, i_{k-1} \rangle$ such that $\langle A[i_0], A[i_1], \ldots, A[i_{k-1}] \rangle$ is alternating.

ϵ-Variant 15.6.2: Define a sequence of numbers $\langle a_0, a_1, \ldots, a_{n-1} \rangle$ to be *weakly alternating* if no three consecutive terms in the sequence are increasing or decreasing. Given an array of numbers A of length n, find a longest subsequence $\langle i_0, \ldots, i_{k-1} \rangle$ such that $\langle A[i_0], A[i_1], \ldots, A[i_{k-1}] \rangle$ is weakly alternating.

ϵ-Variant 15.6.3: Define a sequence of numbers $\langle a_0, a_1, \ldots, a_{n-1} \rangle$ to be *convex* if $a_i < \frac{a_{i-1}+a_{i+1}}{2}$, for $1 \le i \le n-2$. Given an array of numbers A of length n, find a longest subsequence $\langle i_0, \ldots, i_{k-1} \rangle$ such that $\langle A[i_0], A[i_1], \ldots, A[i_{k-1}] \rangle$ is convex.

ϵ-Variant 15.6.4: Define a sequence of numbers $\langle a_0, a_1, \ldots, a_{n-1} \rangle$ to be *bitonic* if there exists k such that $a_i < a_{i+1}$, for $0 \le i < k$ and $a_i > a_{i+1}$, for $k \le i < n-1$. Given an array of numbers A of length n, find a longest subsequence $\langle i_0, \ldots, i_{k-1} \rangle$ such that $\langle A[i_0], A[i_1], \ldots, A[i_{k-1}] \rangle$ is bitonic.

ϵ-Variant 15.6.5: Define a sequence of points in the plane to be *ascending* if each point is above and to the right of the previous point. How would you find a maximum ascending subset of a set of points in the plane?

Problem 15.7, *pg. 119*: *Design an algorithm that takes as input an array A of n numbers and a key k, and returns a longest subarray of A for which the subarray sum is less than or equal to k.*

Solution 15.7: Let R be the prefix sum array for A, i.e., $R[i] = \sum_{k=0}^{i} A[k]$; R can be computed in a single iteration over A. Note that the sum of the elements in the subarray $A[i:j]$ is $R[j] - R[i-1]$ (for convenience, take $R[-1] = 0$).

We can now use the idea of the "efficient frontier" of candidate starting indices: we will never choose i as the starting index if there exists i' such that $i' < i$ and $R[i'] \geq R[i]$. The frontier can be captured by the array $T[i] = \min_{k=i}^{n-1}(R[k])$, which can be computed by a single iteration over R, starting from the end of R and working backwards. Observe that the entries in T are monotonically increasing, i.e., for all $i, T[i] \leq T[i+1]$.

Consider any index i. Let j be the largest index such that $T[j] \leq k + R[i]$. We claim the longest subarray starting at $i+1$ that has a sum less than or equal to k must end at j, inclusive.

Proof:

First note that by definition of T there exists $j' \geq j$ such that $R[j'] = T[j]$. Hence there is a subarray starting at $i+1$ that has length at least $j - i$ that satisfies the subarray sum constraint (i.e., $R[j] - R[i] \leq k$).

Now suppose for contradiction that there exists a longer subarray satisfying the subarray sum constraint starting at $i+1$ and ending at j'' where $j'' > j$. Since this subarray satisfies the subarray sum constraint, we have $R[j''] - R[i] \leq k$, i.e., $R[j''] \leq k + R[i]$. Now j was chosen to be the largest index such that $T[j] \leq k + R[i]$, and by hypothesis $j < j''$ implies $T[j''] \leq R[j''] \leq k + R[i]$. This contradicts the maximality of j, so j is indeed the ending index of the longest subarray starting at i that satisfies the subarray sum constraint.

We have previously shown how T can be computed in $O(n)$ time. Given an index i we can compute the corresponding j using the variant of binary search that finds the greatest entry less than the search key. Specifically, Solution 11.2 on Page 259 shows how to find the smallest element larger than the search key in $O(\log n)$. A symmetric algorithm solves the problem of finding the greatest element less than the search key.

We do this once for each i, and record the longest subarray seen thus far, leading to an overall time complexity of $O(n \log n)$.

```
template <typename T>
pair<int, int> find_longest_subarray_less_equal_k(const vector<T> &A,
                                                  const T &k) {
  // Build the prefix sum according to A
  vector<T> prefix_sum;
  partial_sum(A.cbegin(), A.cend(), back_inserter(prefix_sum));

  vector<T> min_prefix_sum(prefix_sum);
  for (int i = min_prefix_sum.size() - 2; i >= 0; --i) {
    min_prefix_sum[i] = min(min_prefix_sum[i], min_prefix_sum[i + 1]);
  }

  pair<int, int> arr_idx(0, upper_bound(min_prefix_sum.cbegin(),
                                        min_prefix_sum.cend(), k) -
                            min_prefix_sum.cbegin() - 1);
  for (int i = 0; i < prefix_sum.size(); ++i) {
```

```
    int idx = upper_bound(min_prefix_sum.cbegin(), min_prefix_sum.cend(),
                          k + prefix_sum[i]) - min_prefix_sum.cbegin() - 1;
    if (idx - i - 1 > arr_idx.second - arr_idx.first) {
      arr_idx = {i + 1, idx};
    }
  }
  return arr_idx;
}
```

Variant 15.7.1: Design an algorithm for finding the longest subarray of a given array such that the average of the subarray elements is $\leq k$.

Problem 15.8, *pg. 120*: *Let A be an array of n numbers encoding the heights of adjacent buildings of unit width. Design an algorithm to compute the area of the largest rectangle contained in this skyline, i.e., compute* $\max_{i<j}((j-i+1) \times \min_{k=i}^{j} A[k])$.

Solution 15.8: A brute-force approach is to take each (i, j) pair, find the minimum of subarray $A[i:j]$, and multiply that by $j-i+1$. This has time complexity $O(n^3)$, which can improved to $O(n^2)$ by iterating over i and then $j \geq i$ and tracking the minimum height of buildings from i to j, inclusive.

Another approach is to iterate over the buildings in the following fashion: when processing the i-th building, we find the largest rectangle which includes that building and has height at least $A[i]$. Implemented naïvely, this approach has time complexity $O(n^2)$. However, we can improve upon it greatly with the following observation: if $k < k' \leq i$ and $A[k] > A[k']$ then there is no reason to consider Building k, when processing Building i. Therefore we can maintain an "efficient frontier" of candidates from Building 0 to Building i. This consists of a stack of buildings: a building is pushed into the stack iff its height is greater than the height of the building on the top of the stack. Observe that the largest rectangle under Building i that has height $A[i]$ extends out to the last building to the left that is in the stack and has height greater than or equal to $A[i]$. Before adding Building i to the stack, we remove buildings from the stack until the stack is empty or the top building has height less than $A[i]$. A similar stack is used to determine the largest rectangle to the right under Building i.

For each building, it will at most be pushed and popped from each of the two stacks at most once, which leads to an amortized $O(1)$ time complexity per building for stack updates, and an overall $O(n)$ time complexity. Following is the implementation in C++:

```
template <typename T>
T calculate_largest_rectangle(const vector<T> &A) {
  // Calculate L
  stack<int> s;
  vector<int> L;
  for (int i = 0; i < A.size(); ++i) {
    while (!s.empty() && A[s.top()] >= A[i]) {
      s.pop();
    }
    L.emplace_back(s.empty() ? -1 : s.top());
```

```
    s.emplace(i);
  }

  // Clear stack for calculating R
  while (!s.empty()) {
    s.pop();
  }
  vector<int> R(A.size());
  for (int i = A.size() - 1; i >= 0; --i) {
    while (!s.empty() && A[s.top()] >= A[i]) {
      s.pop();
    }
    R[i] = s.empty() ? A.size() : s.top();
    s.emplace(i);
  }

  // For each A[i], find its maximum area include it
  T max_area = 0;
  for (int i = 0; i < A.size(); ++i) {
    max_area = max(max_area, A[i] * (R[i] - L[i] - 1));
  }
  return max_area;
}
```

ϵ-Variant 15.8.1: Find the largest square under the skyline.

Problem 15.9, *pg. 120*: *Let A be an $n \times m$ Boolean 2D array. Design efficient algorithms for the following two problems:*

- *What is the largest 2D subarray containing only 1s?*
- *What is the largest square 2D subarray containing only 1s?*

What are the time and space complexities of your algorithms as a function of n and m?

Solution 15.9: A brute-force approach is to examine all 2D subarrays. Since a 2D subarray is characterized by two diagonally opposite corners the total number of such arrays is $O(m^2n^2)$ Each 2D subarray can be checked by examining the corresponding entries, so the overall complexity is $O(m^3n^3)$. This can be easily reduced to $O(m^2n^2)$ by processing 2D subarrays by size, and reusing results—the 2D subarray $A[i : i + a][j : j + b]$ is feasible iff the 2D subarrays $A[i : i + a - 1][j : j + b - 1]$, $A[i + a : i + a][j : j + b - 1]$, $A[i : i + a - 1][j + b : j + b]$, and $A[i + a : i + a][j + b : j + b]$ are feasible. This is a $O(1)$ time operation, assuming that feasibility of the smaller 2D subarrays has already been computed and stored. (Note that this solution requires $O(m^2n^2)$ storage.)

The following approach lowers the time and space complexity. For each feasible entry $A[i][j]$ we record $(h_{i,j}, w_{i,j})$, where $h_{i,j}$ is the largest L such that all the entries in $A[i : i + L - 1][j : j]$ are feasible, and $w_{i,j}$ is the largest L such that all the entries in $A[i : i][j : j + L - 1]$ are feasible. This computation can be performed in $O(mn)$ time, and requires $O(mn)$ storage.

Now for each feasible entry $A[i][j]$ we calculate the largest 2D subarray that has $A[i][j]$ as its bottom-left corner. We do this by processing each entry in $A[i :$

$i+h_{i,j}-1][j:j]$. As we iterate through the entries in vertical order, we update w to the smallest $w_{i,j}$ amongst the entries processed so far. The largest 2D subarray that has $A[i][j]$ as its bottom-left corner and $A[i'][j]$ as its top-left corner has area $(i'-i+1)w$. We track the largest 2D subarray seen so far across all $A[i][j]$ processed.

The time complexity per $A[i][j]$ is proportional to the number of rows, i.e., $O(n)$, yielding an overall time complexity of $O(mn^2)$, and space complexity of $O(mn)$.

```
class MaxHW {
  public:
    int h, w;
};

int max_rectangle_submatrix(const vector<vector<bool>> &A) {
  // DP table stores (h, w) for each (i, j)
  vector<vector<MaxHW>> table(A.size(), vector<MaxHW>(A.front().size()));

  for (int i = A.size() - 1; i >= 0; --i) {
    for (int j = A[i].size() - 1; j >= 0; --j) {
      // Find the largest h such that (i, j) to (i + h - 1, j) are feasible
      // Find the largest w such that (i, j) to (i, j + w - 1) are feasible
      table[i][j] = A[i][j] ?
                    MaxHW{i + 1 < A.size() ? table[i + 1][j].h + 1 : 1,
                          j + 1 < A[i].size() ? table[i][j + 1].w + 1 : 1} :
                    MaxHW{0, 0};
    }
  }

  int max_rect_area = 0;
  for (int i = 0; i < A.size(); ++i) {
    for (int j = 0; j < A[i].size(); ++j) {
      // Process (i, j) if it is feasible and is possible to update
      // max_rect_area
      if (A[i][j] && table[i][j].w * table[i][j].h > max_rect_area) {
        int min_width = numeric_limits<int>::max();
        for (int a = 0; a < table[i][j].h; ++a) {
          min_width = min(min_width, table[i + a][j].w);
          max_rect_area = max(max_rect_area, min_width * (a + 1));
        }
      }
    }
  }
  return max_rect_area;
}
```

If we are looking for the largest feasible square region, we can improve the complexity as follows—we compute the $(h_{i,j}, w_{i,j})$ values as before. Suppose we know the length s of the largest square region that has $A[i+1][j+1]$ as its bottom-left corner. Then the length of the side of the largest square with $A[i,j]$ as its bottom-left corner is at most $s+1$, which occurs iff $h_{i,j} \geq s+1$ and $w_{i,j} \geq s+1$. The general expression for the length is $\min(s+1, h_{i,j}, w_{i,j})$. Note that this is a $O(1)$ time computation. In total, the run time is $O(mn)$, a factor of n better than before.

The calculations above can be sped up by intelligent pruning. For example, if

we already have a feasible 2D subarray of dimensions $H \times W$, there is no reason to process an entry $A[i, j]$ for which $h_{i,j} \leq H$ and $w_{i,j} \leq W$.

```
class MaxHW {
  public:
    int h, w;
};

int max_square_submatrix(const vector<vector<bool>> &A) {
  // DP table stores (h, w) for each (i, j)
  vector<vector<MaxHW>> table(A.size(), vector<MaxHW>(A.front().size()));

  for (int i = A.size() - 1; i >= 0; --i) {
    for (int j = A[i].size() - 1; j >= 0; --j) {
      // Find the largest h such that (i, j) to (i + h - 1, j) are feasible
      // Find the largest w such that (i, j) to (i, j + w - 1) are feasible
      table[i][j] = A[i][j] ?
                    MaxHW{i + 1 < A.size() ? table[i + 1][j].h + 1 : 1,
                          j + 1 < A[i].size() ? table[i][j + 1].w + 1 : 1} :
                    MaxHW{0, 0};
    }
  }

  // A table stores the length of largest square for each (i, j)
  vector<vector<int>> s(A.size(), vector<int>(A.front().size(), 0));
  int max_square_area = 0;
  for (int i = A.size() - 1; i >= 0; --i) {
    for (int j = A[i].size() - 1; j >= 0; --j) {
      int side = min(table[i][j].h, table[i][j].w);
      if (A[i][j]) {
        // Get the length of largest square with bottom-left corner (i, j)
        if (i + 1 < A.size() && j + 1 < A[i + 1].size()) {
          side = min(s[i + 1][j + 1] + 1, side);
        }
        s[i][j] = side;
        max_square_area = max(max_square_area, side * side);
      }
    }
  }
  return max_square_area;
}
```

The largest 2D subarray can be found in $O(nm)$ time and space using a qualitatively different approach. Essentially, we reduce our problem to n instances of the largest rectangle under the skyline problem described in Problem 15.8 on Page 120. First, for each $A[i][j]$ we determine the largest $h_{i,j}$ such that $A[i : i + h_{i,j} - 1][j : j]$ is feasible. (If $A[i][j] = 0$ then $h_{i,j} = 0$.) The h values can be computed in $O(nm)$ time by using DP. Then for each of the n rows, we compute the largest 2D subarray whose bottom edge is on that row in time $O(m)$, using Solution 15.8 on Page 344. This computation can be performed in time $O(n)$ once the $h_{i,j}$ values have been computed. The final solution is the maximum of the n instances.

```
int max_rectangle_submatrix(const vector<vector<bool>> &A) {
```

```
  vector<vector<int>> table(A.size(), vector<int>(A.front().size()));

  for (int i = A.size() - 1; i >= 0; --i) {
    for (int j = A[i].size() - 1; j >= 0; --j) {
      table[i][j] = A[i][j] ? i + 1 < A.size() ? table[i + 1][j] + 1 : 1 : 0;
    }
  }

  // Find the max among all instances of the largest rectangle
  int max_rect_area = 0;
  for (const vector<int> &t : table) {
    max_rect_area = max(max_rect_area, calculate_largest_rectangle(t));
  }
  return max_rect_area;
}
```

The largest square 2D subarray containing only 1s can be computed similarly, with a minor variant on the algorithm in Solution 15.8 on Page 344.

Problem 15.10, *pg. 120*: *Design an algorithm that takes as arguments a 2D array A and a 1D array S, and determines whether S appears in A. If S appears in A, print the sequence of entries where it appears.*

Solution 15.10: We solve this problem using recursion. The Boolean-valued `matchHelper` function takes A, S, a pair of integers (i, j) encoding an entry in A, and a third integer k encoding the offset into S that needs to be matched from (i, j). The function returns `true` if $k = |S|$ or if $S[k] = A[i][j]$ and an entry adjacent to (i, j) matches S with an offset of $k + 1$.

To avoid repeated calls to the same function with the same argument, we cache results. This reduces the complexity to $O(nml)$, where n and m are the dimensions of A, and l is the length of S—we do a constant amount of work within each call to `match_helper` and the number of calls is bounded by the size of the cache. When we find a match, we print it by printing the arguments to the sequence of calls that led to the success.

```
class HashTuple {
  public:
    size_t operator()(const tuple<int, int, int> &t) const {
      return hash<int>()(get<0>(t)) ^ hash<int>()(get<1>(t)) ^
             hash<int>()(get<2>(t));
    }
};

bool match_helper(const vector<vector<int>> &A, const vector<int> &S,
                  unordered_set<tuple<int, int, int>, HashTuple> cache,
                  int i, int j, int len) {
  if (S.size() == len) {
    return true;
  }

  if (i < 0 || i >= A.size() || j < 0 || j >= A[i].size() ||
      cache.find({i, j, len}) != cache.cend()) {
```

```
    return false;
  }

  if (A[i][j] == S[len] &&
      (match_helper(A, S, cache, i - 1, j, len + 1) ||
       match_helper(A, S, cache, i + 1, j, len + 1) ||
       match_helper(A, S, cache, i, j - 1, len + 1) ||
       match_helper(A, S, cache, i, j + 1, len + 1))) {
    return true;
  }
  cache.insert({i, j, len});
  return false;
}

bool match(const vector<vector<int>> &A, const vector<int> &S) {
  unordered_set<tuple<int, int, int>, HashTuple> cache;
  for (int i = 0; i < A.size(); ++i) {
    for (int j = 0; j < A[i].size(); ++j) {
      if (match_helper(A, S, cache, i, j, 0)) {
        return true;
      }
    }
  }
  return false;
}
```

Problem 15.11, *pg. 120*: *Given two strings, represented as arrays of characters A and B, compute the minimum number of edits needed to transform the first string into the second string.*

Solution 15.11: Let the Levenshtein distance between the two strings A and B be represented by $E(A, B)$. Let's say that a and b are, respectively, the length of strings A and B. We now make two claims:

- If $A[a-1] = B[b-1]$, i.e., the last character of A and B are the same, then $E(A, B) = E(A[0:a-2], B[0:b-2])$. This is because $E(A[0:a-2], B[0:b-2])$ is an upper bound and a lower bound on $E(A, B)$. It is an upper bound, since one way to transform A to B is to transform $A[0:a-2]$ to $B[0:b-2]$. It is a lower bound since we can take a transformation of A to B, and reorder the operations in it to get a transformation of $A[0:a-2]$ into $B[0:b-2]$ that is no longer than the original transformation.
- If $A[a-1] \neq B[b-1]$, i.e., the last two characters of the strings do not match, then

$$E(A, B) = 1 + \min \begin{pmatrix} E(A[0:a-2], B[0:b-2]), \\ E(A[0:a-2], B), \\ E(A, B[0:b-2]) \end{pmatrix}$$

  Clearly, the expression on the right hand side is an upper bound on $E(A, B)$. To show that it is a lower bound, if a smaller sequence transforms A into B, there must be a step where the last character of A becomes the same as the last character of B. This could happen either by inserting a new character at

the end, deleting the last character, or substituting the last character of A with the last character of B. We can reorder the sequence such that this operation happens at the end. The length of the sequence would remain the same and we would still end up with B in the end. If this operation was a "delete", then by deleting this operation, we get a sequence of operations that turn $A[0, a-2]$ into B. If this operation was an "insert", then by dropping this operation, we would have a set of transformations that turn A into $B[0, b-2]$. If this operation was a "substitute", then by discarding this operation, we would have a set of transformations that turn $A[0, a-2]$ into $B[0, b-2]$. In any of those cases, it would be a contradiction if there was a sequence of operations that turned A into B which is smaller than $\min(E(A[0:a-2], B[0:b-2]), E(A[0:a-2], B), E(A, B[0:b-2])) + 1$.

We use the above claims to compute $E(A, B)$. Specifically, we tabulate the values of $E\big(A[0:k], B[0:l]\big)$ for all values of $k < a$ and $l < b$. This takes $O(ab)$ time. We can implement this algorithm using $O(\min(a, b))$ space by reusing space, since we never need more than one row of prior solution at a time. Following is the code in C++.

```
int Levenshtein_distance(string A, string B) {
  // Try to reduce the space usage
  if (A.size() < B.size()) {
    swap(A, B);
  }

  vector<int> D(B.size() + 1);
  // Initialization
  iota(D.begin(), D.end(), 0);

  for (int i = 1; i <= A.size(); ++i) {
    int pre_i_1_j_1 = D[0];  // stores the value of D[i - 1][j - 1]
    D[0] = i;
    for (int j = 1; j <= B.size(); ++j) {
      int pre_i_1_j = D[j];  // stores the value of D[i -1][j]
      D[j] = A[i - 1] == B[j - 1] ?
             pre_i_1_j_1 : 1 + min(pre_i_1_j_1, min(D[j - 1], D[j]));
      // Previous D[i - 1][j] will become the next D[i - 1][j - 1]
      pre_i_1_j_1 = pre_i_1_j;
    }
  }
  return D.back();
}
```

Figure 21.11 on the next page shows the E values for the strings "Carthorse" and "Orchestra". Upper-case and lower-case characters are treated as being different. The Levenshtein distance for this two strings is 8. The longest subsequence which is present in both strings is $\langle r, h, s \rangle$.

ϵ-Variant 15.11.1: Given A and B as above, compute a longest sequence of characters that is a subsequence of A and of B.

		C	a	r	t	h	o	r	s	e
	0	1	2	3	4	5	6	7	8	9
O	1	1	2	3	4	5	6	7	8	9
r	2	2	2	2	3	4	5	6	7	8
c	3	3	3	3	3	4	5	6	7	8
h	4	4	4	4	4	3	4	5	6	7
e	5	5	5	5	5	4	4	5	6	6
s	6	6	6	6	6	5	5	5	5	6
t	7	7	7	7	6	6	6	6	6	6
r	8	8	8	7	7	7	7	6	7	7
a	9	9	8	8	8	8	8	7	7	8

Figure 21.11: The E table for "Carthorse" and "Orchestra".

ϵ-Variant 15.11.2: Given a string A, compute the minimum number of characters you need to delete from A to make the resulting string a palindrome.

Variant 15.11.3: Given a string A and a regular expression r, what is the string in the language of the regular expression r that is closest to A? The distance between strings is the Levenshtein distance specified above.

Problem 15.12, *pg. 121*: *Given a dictionary and a string s, design an efficient algorithm that checks whether s is the concatenation of a sequence of dictionary words. If such a concatenation exists, your algorithm should output it.*

Solution 15.12: This is a straightforward DP problem. If the input string s has length n, we build a table T of length n such that $T[k]$ is a Boolean indicating whether the substring $s(0, k)$ can be decomposed into a sequence of valid words.

We can build a hash table of all the valid words to determine if a string is a valid word in $O(1)$ time. Then $T[k]$ holds iff one of the following two conditions is true:

1. There exists a $j \in [0, k-1]$ such that $T[j]$ is true and $s(j+1, k)$ is a valid word.
2. Substring $s(0, k)$ is a valid word.

This tells us if we can break a given string into valid words, but does not yield the words themselves. We can obtain the words with a little more book-keeping. In table T, along with the Boolean value, we also store the length of the last word in the string.

```
vector<string> word_breaking(const string &s,
                             const unordered_set<string> &dict) {
```

```
  // T[i] stores the length of the last string which composed of s(0, i)
  vector<int> T(s.size(), 0);
  for (int i = 0; i < s.size(); ++i) {
    // Set T[i] if s(0, i) is a valid word
    if (dict.find(s.substr(0, i + 1)) != dict.cend()) {
      T[i] = i + 1;
    }

    // Set T[i] if T[j] != 0 and s(j + 1, i) is a valid word
    for (int j = 0; j < i && T[i] == 0; ++j) {
      if (T[j] != 0 && dict.find(s.substr(j + 1, i - j)) != dict.cend()) {
        T[i] = i - j;
      }
    }
  }

  vector<string> ret;
  // s can be assembled by valid words
  if (T.back()) {
    int idx = s.size() - 1;
    while (idx >= 0) {
      ret.emplace_back(s.substr(idx - T[idx] + 1, T[idx]));
      idx -= T[idx];
    }
    reverse(ret.begin(), ret.end());
  }
  return ret;
}
```

If we want all possible decompositions, we can store all possible values of j that gives us a correct break with each position. However the number of possible decompositions can be exponential here. This is exemplified by the string "itsitsitsits...".

Problem 15.13, *pg. 121*: *Given text, i.e., a string of words separated by single blanks, decompose the text into lines such that no word is split across lines and the messiness of the decomposition is minimized. Each line can hold no more than L characters. How would you change your algorithm if the messiness is the sum of the messinesses of all but the last line?*

Solution 15.13: Let the text W consist of words $\langle w_0, w_1, \ldots, w_{n-1} \rangle$. Let l_k be the length of w_k. Suppose we know the optimum messiness $M(i)$ for each subtext of the form $W_i = \langle w_0, w_1, \ldots, w_i \rangle$. We find the optimum decomposition for $W_{i+1} = \langle w_0, w_1, \ldots, w_{i+1} \rangle$ as follows. Consider the last line. It will be of the form $W_{j:i+1} = \langle w_j, w_{j+1}, \ldots, w_{i+1} \rangle$, where j could be $i+1$ (word w_{i+1} lies on a line by itself). The optimum messiness for a decomposition is then $M(j-1) + 2^{L - l_{i+1} - \sum_{k=j}^{i}(l_k+1)}$.

We can compute an optimum messiness for W_{i+1} by iterating from $j = i+1$ down to the first f such that $l_{i+1} + \sum_{k=f}^{i}(l_k + 1) > L$. For each value of j we perform constant work to compute the optimum messiness, assuming we form $\sum_{k=j}^{i}(l_k + 1)$ incrementally. Since each line is constrained to L characters, we will not examine more than L words for each word processed, leading to an $O(nL)$ time complexity. Naïvely, the space complexity is $O(n)$ for storing M; however, since we never examine

more than L previous words on the line that w_{i+1} is on, we can reuse space and reduce the additional storage to $O(L)$.

Now we consider the case where the messiness of a decomposition does not include the messiness of the final line. First, we compute M as above. If the final line is $\langle w_j, w_{j+1}, \ldots, w_{n-1} \rangle$, the optimum messiness will be $M(j-1)$. Not more than L (actually $\lfloor \frac{L}{2} \rfloor + 1$) possibilities exist for the final line, and we can compute the optimum messiness once M is computed by considering all the possibilities of the final line. The time and space bounds are the same as before.

```
int find_pretty_printing(const vector<string> &W, const int &L) {
  // Calculate M(i)
  vector<long> M(W.size(), numeric_limits<long>::max());
  for (int i = 0; i < W.size(); ++i) {
    int b_len = L - W[i].size();
    M[i] = min((i - 1 < 0 ? 0 : M[i - 1]) + (1 << b_len), M[i]);
    for (int j = i - 1; j >= 0; --j) {
      b_len -= (W[j].size() + 1);
      if (b_len < 0) {
        break;
      }
      M[i] = min((j - 1 < 0 ? 0 : M[j - 1]) + (1 << b_len), M[i]);
    }
  }

  // Find the minimum cost without considering the last line
  long min_mess = (W.size() >= 2 ? M[W.size() - 2] : 0);
  int b_len = L - W.back().size();
  for (int i = W.size() - 2; i >= 0; --i) {
    b_len -= (W[i].size() + 1);
    if (b_len < 0) {
      return min_mess;
    }
    min_mess = min(min_mess, (i - 1 < 0 ? 0 : M[i - 1]));
  }
  return min_mess;
}
```

Variant 15.13.1: Suppose the messiness of a line ending with b blank characters is defined to be b. Can you solve the messiness minimization problem in $O(n)$ time and $O(1)$ space?

Problem 15.14, *pg. 122*: *Design an efficient algorithm for computing $\binom{n}{k}$ which has the property that it never overflows if $\binom{n}{k}$ can be represented as a 32-bit integer; assume n and k are integers.*

Solution 15.14: It is tempting to proceed by pairing terms in the numerator and denominator for the expression for $\binom{n}{k}$ that have common factors and cancel them out. This approach is unsatisfactory because of the need to have factorizations.

The binomial coefficients satisfy several identities, the most basic of which is the

addition formula:

$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$$

Various proofs exist of this identity, ranging from the combinatorial interpretation to induction and, finally, direct manipulation of the expressions.

This identity yields a straightforward recursion for $\binom{n}{k}$. The base cases are $\binom{r}{r}$ and $\binom{r}{0}$, both of which are 1. The individual results from the subcalls are 32-bit integers and if $\binom{n}{k}$ can be represented by a 32-bit integer, they can too; so, overflow is not a concern.

The recursion can lead to repeated subcalls and consequently exponential run times, which can be avoided by caching intermediate results as in DP. The number of subproblems is $O(n^2)$ and the results can be combined in $O(1)$ time, yielding an $O(n^2)$ complexity bound.

```
int compute_binomial_coefficients(const int &n, const int &k) {
  vector<vector<int>> table(n + 1, vector<int>(k + 1));
  // Basic case: C(i, 0) = 1
  for (int i = 0; i <= n; ++i) {
    table[i][0] = 1;
  }
  // Basic case: C(i, i) = 1
  for (int i = 1; i <= k; ++i) {
    table[i][i] = 1;
  }

  // C(i, j) = C(i - 1, j) + C(i - 1, j - 1)
  for (int i = 2; i <= n; ++i) {
    for (int j = 1; j < i && j <= k; ++j) {
      table[i][j] = table[i - 1][j] + table[i - 1][j - 1];
    }
  }
  return table[n][k];
}
```

Problem 15.15, *pg. 122*: *You have an aggregate score s and W which specifies the points that can be scored in an individual play. How would you find the number of combinations of plays that result in an aggregate score of s? How would you compute the number of distinct sequences of individual plays that result in a score of s?*

Solution 15.15: Let $W = \{w_0, w_1, \ldots, w_{n-1}\}$ be the possible scores for individual plays. Let X be the set $\{\langle x_0, x_1, \ldots, x_{n-1}\rangle \mid \sum_{i=0}^{n-1} w_i x_i = s\}$. We want to compute $|X|$. Observe that x_0 can take any value in $[0, \lfloor \frac{s}{w_0} \rfloor]$. Therefore, we can partition X into subsets of vectors of the form $\{\langle x_0, x_1, \ldots, x_{n-1}\rangle\}$, where $0 \le x_0 \le \lfloor \frac{s}{w_0} \rfloor$. We can determine the size of each of these subsets by solving the same problem in one fewer dimension—specifically for each x_0 we count the number of combinations in which $s - x_0 w_0$ can be achieved using plays $\{w_1, w_2, \ldots, w_{n-1}\}$. The base case corresponds to computing the number of ways in which a score $t \le s$ can be formed with the w_{n-1}-score plays, which is 1 or 0, depending on whether w_{n-1} evenly divides t.

The algorithm outlined above has exponential complexity. We can use DP to reduce its complexity—for each $t \leq s$ and $d \in [1, n-1]$ we cache the number of combinations of ways in which $w_d, \ldots, w_{n-1}$ can be used to achieve t. By iterating first over W and then over t, we can reuse space. This is the approach given below.

```
int count_combinations(const int &k, const vector<int> &score_ways) {
  vector<int> combinations(k + 1, 0);
  combinations[0] = 1;  // 1 way to reach 0
  for (const int &score : score_ways) {
    for (int j = score; j <= k; ++j) {
      combinations[j] += combinations[j - score];
    }
  }
  return combinations[k];
}
```

We can compute the number of permutations of scores which lead to an aggregate score of s using recursion. Suppose we know for all $u < v$ the number of permutations of ways in which u can be achieved. We can achieve v points by first scoring $v - w_i$ points followed by w_i. Observe each of these is a distinct permutation. The recursion can be converted to DP by caching the number of permutations yielding t for each $t < s$.

```
int count_permutations(const int &k, const vector<int> &score_ways) {
  vector<int> permutations(k + 1, 0);
  permutations[0] = 1;  // 1 way to reach 0
  for (int i = 0; i <= k; ++i) {
    for (const int &score : score_ways) {
      if (i >= score) {
        permutations[i] += permutations[i - score];
      }
    }
  }
  return permutations[k];
}
```

Variant 15.15.1: Suppose the final score is given in the form (s, s'), i.e., Team 1 scored s points and Team 2 scored s' points. How would you compute the number of distinct scoring sequences which result in this score? For example, if the final score is $(6, 3)$ then Team 1 scores 3, Team 2 scores 3, Team 1 scores 3 is a scoring sequence which results in this score.

Variant 15.15.2: Suppose the final score is (s, s'). How would you compute the maximum number of times the team that lead could have changed? For example, if $s = 10$ and $s' = 6$, the lead could have changed 4 times: Team 1 scores 2, then Team 2 scores 3 (lead change), then Team 1 scores 2 (lead change), then Team 2 scores 3 (lead change), then Team 1 scores 3 (lead change) followed by 3.

Problem 15.16, *pg. 122*: *How many ways can you go from the top-left to the bottom-right in an $n \times m$ 2D array? How would you count the number of ways in the presence of obstacles, specified by an $n \times m$ Boolean 2D array B, where a* `true` *represents an obstacle.*

Solution 15.16: This problem can be solved using a straightforward application of DP: the number of ways to get to (i, j) is the number of ways to get to $(i-1, j)$ plus the number of ways to get to $(i, j-1)$. (If $i = 0$ or $j = 0$, there is only one way to get to (i, j).) The matrix storing the number of ways to get to (i, j) for the configuration in Figure 15.8 on Page 122 is show in Figure 21.12.

```
int number_of_ways(const int &n, const int &m) {
  vector<vector<int>> A(n, vector<int>(m, 0));
  A[0][0] = 1;  // 1 way to start from (0, 0)
  for (int i = 0; i < n; ++i) {
    for (int j = 0; j < m; ++j) {
      A[i][j] += (i < 1 ? 0 : A[i - 1][j]) + (j < 1 ? 0 : A[i][j - 1]);
    }
  }
  return A.back().back();
}
```

We can improve on the above by noting that we do not need an $n \times m$ 2D array, since to fill in the i-th row we do not need values from rows before $i-1$.

1	1	1	1	1
1	2	3	4	5
1	3	6	10	15
1	4	10	20	35
1	5	15	35	70

Figure 21.12: The number of ways to get from $(0, 0)$ to (i, j) for $0 \le i, j \le 4$.

An even better solution is based on the fact that each path from $(0, 0)$ to $(n-1, m-1)$ is a sequence of $m-1$ horizontal steps and $n-1$ vertical steps. There are $\binom{n+m-2}{n-1} = \binom{n+m-2}{m-1} = \frac{(n+m-2)!}{(n-1)!(m-1)!}$ such paths. This value can be efficiently computed without division by using DP; refer to Solution 15.14 on Page 353 for details.

Our first solution generalizes trivially to obstacles: if there is an obstacle at (i, j) there are zero ways of getting from $(0, 0)$ to (i, j).

```
// Given the dimensions of A, n and m, and B, return the number of ways
// from A[0][0] to A[n - 1][m - 1] considering obstacles
int number_of_ways_with_obstacles(const int &n, const int &m,
                                  const vector<vector<bool>> &B) {
  vector<vector<int>> A(n, vector<int>(m, 0));
  // No way to start from (0, 0) if B[0][0] == true
  A[0][0] = !B[0][0];
  for (int i = 0; i < n; ++i) {
    for (int j = 0; j < m; ++j) {
      if (B[i][j] == 0) {
        A[i][j] += (i < 1 ? 0 : A[i - 1][j]) + (j < 1 ? 0 : A[i][j - 1]);
```

```
      }
    }
  }
  return A.back().back();
}
```

Variant 15.16.1: A decimal number is a sequence of digits, i.e., a sequence over $\{0, 1, 2, \ldots, 9\}$. The sequence has to be of length 1 or more, and the first element in the sequence cannot be 0. Call a decimal number D *monotone* if $D[i] \leq D[i+1], 0 \leq i < |D|$. Write a function which takes as input a positive integer k and computes the number of decimal numbers of length k that are monotone.

Variant 15.16.2: Call a decimal number D, as defined above, *strictly monotone* if $D[i] < D[i+1], 0 \leq i < |D|$. Write a function which takes as input a positive integer k and computes the number of decimal numbers of length k that are strictly monotone.

Problem 15.17, *pg. 123*: *Write a program that computes the maximum value of fish a fisherman can catch on a path from the upper leftmost point to the lower rightmost point. The fisherman can only move down or right, as illustrated in Figure 15.9 on Page 123.*

Solution 15.17: The maximum value that the fisherman can obtain while getting to (i, j) is the larger of the maximum value he can catch on his way to $(i-1, j)$ or $(i, j-1)$ plus the value of the fish at (i, j). This is the basis for the DP algorithm shown below. (If $i = 0$ or $j = 0$, there is a unique path to (i, j) from $(0, 0)$.)

```
template <typename T>
T maximize_fishing(vector<vector<T>> A) {
  for (int i = 0; i < A.size(); ++i) {
    for (int j = 0; j < A[i].size(); ++j) {
      A[i][j] += max(i < 1 ? 0 : A[i - 1][j], j < 1 ? 0 : A[i][j - 1]);
    }
  }
  return A.back().back();
}
```

***e*-Variant 15.17.1**: Solve the same problem when the fisherman can begin and end at any point. He must still move down or right. (Note that the value at (i, j) may be negative.)

Problem 15.18, *pg. 123*: *Design an efficient algorithm for computing the maximum margin of victory for the starting player in the pick-up-coins game.*

Solution 15.18: First we note that maximizing the margin of victory for Player F is the same as maximizing the value of the coins picked up by Player F—this follows from the fact that the sum of the coins picked by the two players is equal to the total sum of all the coins.

Call the sum of the coins selected by a player his revenue. Let $f(a,b)$ be the maximum revenue a player can get when it is his turn to play, and the coins remaining on the table are at indices a to b, inclusive, where $a \leq b$. Then $f(a,b)$ satisfies the following equations:

$$f(a,b) = \begin{cases} \max\left(\begin{array}{l} C[a] + \min\left(\begin{array}{l} f(a+2,b), \\ f(a+1,b-1) \end{array}\right), \\ C[b] + \min\left(\begin{array}{l} f(a+1,b-1), \\ f(a,b-2) \end{array}\right) \end{array}\right), & \text{if } a \leq b; \\ 0, & \text{otherwise.} \end{cases}$$

The logic is that the second player will choose the coin that maximizes his profit, which is equivalent to minimizing the profit that the first player will make after the second player makes his move.

We can solve for f using DP—there are $\frac{n(n+1)}{2}$ possible arguments for $f(a,b)$ where n is the number of coins, and the work required to compute f from previously computed values is constant. Hence f can be computed in $O(n^2)$ time.

```
template <typename CoinType>
CoinType pick_up_coins_helper(const vector<CoinType> &C, const int &a,
                              const int &b, vector<vector<CoinType>> &T) {
  if (a > b) {
    return 0;  // base condition
  }

  if (T[a][b] == -1) {
    T[a][b] = max(C[a] + min(pick_up_coins_helper(C, a + 2, b, T),
                             pick_up_coins_helper(C, a + 1, b - 1, T)),
                  C[b] + min(pick_up_coins_helper(C, a + 1, b - 1, T),
                             pick_up_coins_helper(C, a, b - 2, T)));
  }
  return T[a][b];
}

template <typename CoinType>
CoinType pick_up_coins(vector<CoinType> &C) {
  vector<vector<CoinType>> T(C.size(), vector<int>(C.size(), -1));
  return pick_up_coins_helper(C, 0, C.size() - 1, T);
}
```

The DP algorithm applied to the configuration in Figure 4.6 on Page 44, shows a maximum gain for F is 140¢, i.e., whatever strategy S plays, F can guarantee a gain of at least 140¢.

ϵ-Variant 15.18.1: You are given two fixed arrays of numbers A and B, each of length k, and another array C, also of length k. You can assign $C[i]$ to one of $A[i]$, $B[i]$, or 0, subject to the constraint that if $C[i] = A[i]$ then $C[i-1]$ must be assigned 0 if $i > 0$. Design an algorithm that computes an assignment to C that maximizes the sum of the elements in C.

Problem 15.19, *pg. 123*: *Design an algorithm for minimizing power that takes as input a rooted tree and assigns each node to a low or high voltage, subject to the design constraint.*

Solution 15.19: Let $l(r)$ and $h(r)$ be the power consumption of node r under low and high voltages, respectively. Let $L(r)$ be the minimum possible power that can be achieved when we assign a low voltage to r. Let $H(r)$ be the minimum possible power that can be achieved when r is assigned a high voltage.

Denote the set of all nodes that are inputs to r by $I(r)$. Then the following recurrence relationships must hold for L and H:

$$\begin{aligned} L(r) &= l(r) + \sum_{c \in I(r)} H(c), \\ H(r) &= h(r) + \sum_{c \in I(r)} \min\big(L(c), H(c)\big). \end{aligned}$$

Using these equations, we can tabulate the values of L and H for all nodes. The desired solution is the minimum of the values of L and H for the root of the tree. Since we do a constant number of operations per node, the overall complexity is $O(n)$, where n is the number of nodes.

Problem 15.20, *pg. 124*: *Implement cutpoint selection to minimize the number of nodes in the two-dimensional tree representing an image.*

Solution 15.20: We maintain a cache mapping each subrectangle of the image to the minimum number of nodes needed when representing it as a two-dimensional tree. Computing a cache entry entails first checking if the corresponding image is monochromatic. If it is not monochromatic the algorithm tries all possible choices for the cutpoint. Given a cutpoint, the optimum tree is determined by making four lookups into the cache.

The cache has no more than $\binom{(m+1)\times(n+1)}{2} = O(n^2m^2)$ entries, which is an upper bound on the number of recursive calls. The time spent within a call is dominated by iterating through the different choices of the cutpoint, i.e., $O(mn)$, leading to a $O(m^3n^3)$ time complexity for the entire algorithm.

If large monochromatic regions are present in the image, the run time can be sped up by computing the number of 1s in $P[0:i, 0:j]$, for $0 \le i < m, 0 \le j < n$. Call this quantity $N[i,j]$. It can be computed in $O(nm)$ time by iterating through rows in ascending order. Precomputing N accelerates checking subrectangles in P for monochromaticity: the number of 1s in $P[i:i', j:j']$ is $N[i',j'] - N[i-1,j'] - N[i',j-1] + N[i-1,j-1]$.

```
class Point {
  public:
    int i, j;

    const bool operator>(const Point &that) const {
      return i > that.i || j > that.j;
    }

```

```
    // Equal function for hash
    const bool operator==(const Point &that) const {
      return i == that.i && j == that.j;
    }
};

// Hash function for Point
class HashPoint {
  public:
    const size_t operator()(const Point &p) const {
      return hash<int>()(p.i) ^ hash<int>()(p.j);
    }
};

class TreeNode {
  public:
    int node_num;  // stores the number of node in its subtree

    Point lowerLeft, upperRight;

    // Store the SW, NW, NE, and SE rectangles if color is mixed
    vector<shared_ptr<TreeNode>> children;
};

bool is_monochromatic(const vector<vector<int>> &image_sum,
                      const Point &lower_left, const Point &upper_right) {
  int pixel_sum = image_sum[upper_right.i][upper_right.j];
  if (lower_left.i >= 1) {
    pixel_sum -= image_sum[lower_left.i - 1][upper_right.j];
  }
  if (lower_left.j >= 1) {
    pixel_sum -= image_sum[upper_right.i][lower_left.j - 1];
  }
  if (lower_left.i >= 1 && lower_left.j >= 1) {
    pixel_sum += image_sum[lower_left.i - 1][lower_left.j - 1];
  }
  return pixel_sum == 0 ||  // totally white
         pixel_sum == (upper_right.i - lower_left.i + 1) *  // totally black
                      (upper_right.j - lower_left.j + 1);
}

shared_ptr<TreeNode> calculate_optimal_2D_tree_helper(
    const vector<vector<int>> &image, const vector<vector<int>> &image_sum,
    const Point &lower_left, const Point &upper_right,
    unordered_map<Point,
                  unordered_map<Point, shared_ptr<TreeNode>, HashPoint>,
                  HashPoint> &table) {
  // Illegal rectangle region, returns empty node
  if (lower_left > upper_right) {
    return shared_ptr<TreeNode>(new TreeNode{0, lower_left, upper_right});
  }

  if (table[lower_left].find(upper_right) == table[lower_left].cend()) {
    if (is_monochromatic(image_sum, lower_left, upper_right)) {
      shared_ptr<TreeNode> p(new TreeNode{1, lower_left, upper_right});
```

```
      table[lower_left][upper_right] = p;
    } else {
      shared_ptr<TreeNode>
        p(new TreeNode{numeric_limits<int>::max(), lower_left, upper_right});
      for (int s = lower_left.i; s <= upper_right.i + 1; ++s) {
        for (int t = lower_left.j; t <= upper_right.j + 1; ++t) {
          if ((s != lower_left.i && s != upper_right.i + 1) ||
              (t != lower_left.j && t != upper_right.j + 1)) {
            vector<shared_ptr<TreeNode>> children = {
              // SW rectangle
              calculate_optimal_2D_tree_helper(image, image_sum, lower_left,
                                               Point{s - 1, t - 1}, table),
              // NW rectangle
              calculate_optimal_2D_tree_helper(image, image_sum,
                                               Point{lower_left.i, t},
                                               Point{s - 1, upper_right.j},
                                               table),
              // NE rectangle
              calculate_optimal_2D_tree_helper(image, image_sum, Point{s, t},
                                               upper_right, table),
              // SE rectangle
              calculate_optimal_2D_tree_helper(image, image_sum,
                                               Point{s, lower_left.j},
                                               Point{upper_right.i, t - 1},
                                               table)};

            int node_num = 1;  // itself
            for (shared_ptr<TreeNode> &child : children) {
              node_num += child->node_num;
              // Remove the child contains no node
              if (child->node_num == 0) {
                child = nullptr;
              }
            }
            if (node_num < p->node_num) {
              p->node_num = node_num, p->children = children;
            }
          }
        }
      }
      table[lower_left][upper_right] = p;
    }
  }
  return table[lower_left][upper_right];
}

shared_ptr<TreeNode> calculate_optimal_2D_tree(
    const vector<vector<int>> &image) {
  vector<vector<int>> image_sum(image);
  for (int i = 0; i < image.size(); ++i) {
    partial_sum(image_sum[i].cbegin(), image_sum[i].cend(),
                image_sum[i].begin());
    for (int j = 0; i > 0 && j < image[i].size(); ++j) {
      image_sum[i][j] += image_sum[i - 1][j];
    }
```

```
  }

  unordered_map<Point,
                unordered_map<Point, shared_ptr<TreeNode>, HashPoint>,
                HashPoint> table;
  return calculate_optimal_2D_tree_helper(image, image_sum, Point{0, 0},
                                          Point{static_cast<int>(
                                            image.size() - 1),
                                            static_cast<int>(
                                            image[0].size() - 1)},
                                          table);
}
```

Variant 15.20.1: Define the intersection of two images that have the same dimension to be the image that is white wherever either image is white and black otherwise. Write a function that takes two two-dimensional trees representing images, and returns a two-dimensional tree that represents their intersection.

Problem 15.21, *pg. 125*: *Given n queries, compute an order in which to process queries that minimizes the total waiting time.*

Solution 15.21: Consider a schedule in which the i-th client is the c_i-th one to be processed; the j-th client processed corresponds to client with ID d_j. Then the waiting time for the i-th client is $\sum_{j<c_i} t_{d_j}$. Hence sum of all the wait times would be

$$\sum_{i=1}^{n} T_i = \sum_{i=1}^{n} \sum_{j<c_i} t_{d_j} = \sum_{i=1}^{n} t_i(n - c_i).$$

Since we want to minimize the total wait time for all the queries and each c_i takes a value between 1 and n, it follows that the queries that take the smallest time must get served first. Hence we should sort the queries by their service time and then process them in the order of non-decreasing service time.

```
template <typename T>
T minimum_waiting_time(vector<T> service_time) {
  // Sort the query time in increasing order
  sort(service_time.begin(), service_time.end());

  T waiting = 0;
  for (int i = 0; i < service_time.size(); ++i) {
    waiting += service_time[i] * (service_time.size() - (i + 1));
  }
  return waiting;
}
```

Problem 15.22, *pg. 125*: *Design an algorithm that computes the least number of tutors needed to schedule a set of requests.*

Solution 15.22: We schedule tutors greedily: as soon as there is a request that cannot be handled by the previously assigned tutors, we choose a new tutor.

This scheme is simple to implement, but it is not completely trivial to prove that it is optimum.

We will use the notion of slack to prove optimality. Suppose the last tutor scheduled begins at time t, and he completes the last lesson he was assigned at time t'. Then the *slack* in the schedule is defined to be $t + 120 - t'$ minutes.

Consider a set of requests $j_1, \ldots, j_n$ such that the requests are ordered by the time they need to be completed. We claim that greedy scheduling is optimum.

Proof:

We use induction on the number of requests. For our induction hypothesis, in addition to the claim that the number of tutors is minimized, we also claim that the schedule maximizes the slack. For $n = 1$, the greedy algorithm sends exactly one tutor at the start time of the request. Clearly this is the strategy that uses the minimum number of tutors and no more slack is possible.

Assume the induction hypothesis holds for all $n \leq k$. We will now prove it for $n = k+1$. Consider the requests $j_1, \ldots, j_k$ sorted by their start time. When the next request j_{k+1} is added to the list, either it can be covered by an existing tutor, which is determined by the slack, or it may require a new tutor.

If the new request can be covered by the slack, using an existing tutor must be optimum with respect to the number of tutors. If we needed at least m tutors to cover the first k requests, we cannot cover the $k + 1$ requests with fewer tutors. Also, since the schedule for the first k requests maximized the slack, we cannot have a schedule with m tutors that covers all $k + 1$ requests and has more slack.

If an additional tutor is needed for the $k + 1$-th request, it must be that the m-th tutor did not have the slack to cover the last request. If there exists another way to cover the requests with m or less tutors, then we can use the same set of tutors to cover the first k requests and get a bigger slack, which contradicts the induction hypothesis. Since the $(m+1)$-th tutor will start exactly when the last request starts, this schedule is slack maximizing.

Problem 15.23, *pg. 126*: *Design an algorithm that takes as input a pair of arrays specifying jobs per task and server capacities, and returns an assignment of jobs to servers for which all tasks complete within one unit time. No server may process more than one job for a given task. If no such assignment exists, your algorithm should indicate that.*

Solution 15.23: Let X be an $m \times n$ Boolean 2D array. Define X to *realize* (T, S) if for all i, $T[i] = \sum_{j=0}^{n-1} X[i, j]$ and for all j, $S[j] \geq \sum_{i=0}^{m-1} X[i, j]$. Clearly X can be viewed as an assignment of jobs to servers; since X is Boolean, we never have more than one job from the same task assigned to the same server.

Let s be the index of a maximal element S, i.e., a server with maximum capacity. We claim that (T, S) is realizable iff it can be realized by a 2D array X in which server s has the greatest load, i.e., Column s has the maximum number of 1s out of all the columns of X.

Proof:

If in some legal assignment the server s is not a most loaded server, look at a server s' that is most loaded. There must exist a task t which utilizes s' but not s, since otherwise s would be a most loaded server. We can then move that job from t to s without violating any constraints. (Note that s must have residual capacity, since otherwise it would not be the server with maximum capacity.)

The above reasoning can be used to develop an algorithm for building the 2D array X that implements the assignment, if a valid assignment exists: initialize X to all 0s. Sort servers in decreasing order of capacity. Starting with a server with maximum capacity, greedily assign jobs from tasks to that server. Iterate until there all tasks have been assigned or no servers remain. In the latter case, the above theorem guarantees that no assignment of jobs to servers satisfying the constraints.

In the code listed below, we use a straightforward implementation of the greedy algorithm. Specifically, we iterate over the servers in an outer loop and tasks in an inner loop. The time complexity is $O(nm)$. The code includes some heuristics for early detection of infeasibility, such as a single task requiring more servers than are available, or the aggregated tasks exceeding the total available server capacity.

```
const bool comp(const pair<int, int> &a, const pair<int, int> &b) {
  return a.second > b.second;
}

vector<vector<bool>> find_feasible_job_assignment(const vector<int> &T,
                                                  const vector<int> &S) {

  int T_total = accumulate(T.cbegin(), T.cend(), 0),  // aggregated work units
      S_total = accumulate(S.cbegin(), S.cend(), 0,
                           [&T](const int &x, const int &y) -> int {
                             return x + min(y, static_cast<int>(T.size()));
                           });  // tighter bound of server capacity
  if (T_total > S_total || *max_element(T.cbegin(), T.cend()) > S.size()) {
    return {};  // too many jobs or one task needs too many servers
  }

  vector<pair<int, int>> T_idx_data, S_idx_data;
  for (int i = 0; i < T.size(); ++i) {
    T_idx_data.emplace_back(i, T[i]);
  }
  for (int j = 0; j < S.size(); ++j) {
    S_idx_data.emplace_back(j, S[j]);
  }

  sort(S_idx_data.begin(), S_idx_data.end(), comp);
  vector<vector<bool>> X(T.size(), vector<bool>(S.size(), false));
  for (int j = 0; j < S_idx_data.size(); ++j) {
    if (S_idx_data[j].second < T_idx_data.size()) {
      nth_element(T_idx_data.begin(),
                  T_idx_data.begin() + S_idx_data[j].second, T_idx_data.end(),
                  comp);
    }

    // Greedily assign jobs
```

```
    int size = min(static_cast<int>(T_idx_data.size()), S_idx_data[j].second);
    for (int i = 0; i < size; ++i) {
      if (T_idx_data[i].second) {
        X[T_idx_data[i].first][S_idx_data[j].first] = true;
        --T_idx_data[i].second;
        --T_total;
      }
    }
  }
  if (T_total) {
    return {};  // still some jobs remain, no feasible assignment
  }
  return X;
}
```

Problem 15.24, *pg. 126*: *You have n users with unique hash codes h_0 through h_{n-1}, and m servers. The hash codes are ordered by index, i.e., $h_i < h_{i+1}$ for $i \in [0, n-2]$. User i requires b_i bytes of storage. The values $k_0 < k_1 < \cdots < k_{m-2}$ are used to assign users to servers. Specifically, the user with hash code c gets assigned to the server with the lowest ID i such that $c \leq k_i$, or to server $m-1$ if no such i exists. The load on a server is the sum of the bytes of storage of all users assigned to that server. Compute values for $k_0, k_1, \ldots, k_{m-1}$ that minimizes the load on the most heavily loaded server.*

Solution 15.24: Let $L(p, q)$ be the maximum load on a server when users with hash codes h_0 through h_p are assigned to Servers 0 through q in an optimum way, i.e., when the maximum load is minimized. Then following recurrence holds:

$$L(p,q) = \min_{x \in [0,p]} \left(\max \left(L(x, q-1), \sum_{i=x+1}^{p} b_i \right) \right)$$

In other words, to find the optimum assignment of users with hash codes $\{h_0, h_1, \ldots, h_p\}$ to q servers, we find x such that if we assign the first $x+1$ users optimally to $q-1$ servers and the remainder to Server q, the maximum load on a given server is minimized.

We can use the recurrence to tabulate the values in L till we get $L(n-1, m-1)$. The base case corresponds to entries of the form $L(p, 0)$, in which case the maximum load is $\sum_{i=0}^{p} b_i$. The time complexity to compute each $L(i, j)$ is $O(n)$, so the overall complexity to compute $L(n-1, m-1)$ is $O(n^2 m)$.

A qualitatively different approach, based on the greedy method, is to check whether $k_0, k_1, \ldots, k_{m-1}$ can be chosen so as to ensure that no server stores more than b bytes. For a given b, this can easily be done—iterate through the n users in the order of their hash codes, and assign them to the servers greedily, i.e., assign users to servers, moving on the next server when the capacity of the current server is exceeded. We can perform binary search to get the minimum b, and the corresponding values for $k_0, k_1, \ldots, k_{m-1}$. The time complexity of the approach is $O(n \log W)$, where W is the total number of bytes that are to be stored, i.e., $W = \sum_{i=0}^{m-1} b_i$.

This approach is much faster in practice: when $n = 10000$, $m = 100$, and loads are uniform integer random variables in the range $[1, 100]$, the $O(n^2m)$ DP algorithm takes over an hour on our machine. In contrast, binary search for w took 0.1 seconds. Furthermore, binary search requires no additional storage beyond that needed to store the final result. The complexity of the code is also greatly reduced compared to the DP algorithm.

```
bool greedy_assignment(const vector<int> &user_file_size,
                       const int &server_num, const int &limit,
                       vector<int> &assign_res) {
  int server_idx = 0;
  for (const int &file : user_file_size) {
    while (server_idx < server_num && file + assign_res[server_idx] > limit) {
      ++server_idx;
    }

    if (server_idx >= server_num) {
      return false;
    } else {
      assign_res[server_idx] += file;
    }
  }
  return true;
}

vector<int> decide_load_balancing(vector<int> user_file_size,
                                  const int &server_num) {
  // Uses binary search to find the assignment with minimized maximum load
  int l = 0,
      r = accumulate(user_file_size.cbegin(), user_file_size.cend(), 0);
  vector<int> feasible_assignment;
  while (l <= r) {
    int m = l + ((r - l) >> 1);
    vector<int> assign_res(server_num, 0);
    bool is_feasible = greedy_assignment(user_file_size, server_num, m,
                                         assign_res);
    if (is_feasible) {
      feasible_assignment = assign_res;
      r = m - 1;
    } else {
      l = m + 1;
    }
  }
  return feasible_assignment;
}
```

Problem 15.25, *pg. 126*: *Implement first-fit to run in $O(n \log n)$ time.*

Solution 15.25: This can be trivially done in $O(n^2)$ time if we do a linear scan through the boxes for each new object to find the first box where it would fit.

To speed things up, we maintain a *"tournament tree"* data structure which captures remaining capacities as well as the box sequence. The tournament tree is organized

as a complete binary tree. Each leaf corresponds to a box, and the leaf order from left-to-right is the box sequence.

For simplicity, we assume $n = 2^k$ for some k. We start with as many boxes as items. For each internal node v, we record the largest remaining capacity $v.max$ that exists amongst the boxes corresponding to leaves at the subtree rooted at v. Finding the first box that has capacity c can be done recursively. Let r be the root: if $r.left.max \geq c$, we search the root's left child, otherwise, we must use a box on the right side. (Note that $r.max \geq c$ always holds, since we have n boxes and n items.)

After an item is placed in a box, the remaining capacity of the box changes. The only internal nodes v whose $v.max$ changes are those that are ancestors of the leaf corresponding to that box. Updating these nodes consists of simply updating their $v.max$ to the maximum of the $v.max$ of their children in a bottom-up order.

A complete binary tree on n leaves has exactly $n-1$ internal nodes. The height of a complete binary tree on $2n-1$ nodes is $\lceil \log(2n-1) \rceil$, implying the update runs in $O(\log n)$ time.

Building the initial tournament tree has time complexity $O(n)$. Each item takes $O(\log n)$ time to process, leading to the desired $O(n \log n)$ time bound.

```
template <typename ItemType, typename CapacityType>
class TournamentTree {
  private:
    class TreeNode {
      public:
        CapacityType cap;  // leaf: remaining capacity in the box
                           // non-leaf: max remaining capacity in the subtree
        vector<ItemType> items;  // stores the items in the leaf node
    };

    // Store the complete binary tree. For tree[i],
    // left subtree is tree[2i + 1], and right subtree is tree[2i + 2].
    vector<TreeNode> tree;

    // Recursively inserts item in tournament tree
    void insertHelper(const int &idx, const ItemType &item,
                      const CapacityType &cap) {
      int left = (idx << 1) + 1, right = (idx << 1) + 2;
      if (left < tree.size()) {  // internal node
        insertHelper(tree[left].cap >= cap ? left : right, item, cap);
        tree[idx].cap = max(tree[left].cap, tree[right].cap);
      } else {  // leaf node
        tree[idx].cap -= cap, tree[idx].items.emplace_back(item);
      }
    }

  public:
    // n items, and each box has unit_cap
    TournamentTree(int n, const CapacityType &unit_cap) :
      // Complete tree with n leafs has 2n - 1 nodes
      tree(vector<TreeNode>((n << 1) - 1, {unit_cap})) {}

    void insert(const ItemType &item, const CapacityType &item_cap) {
```

```
        insertHelper(0, item, item_cap);
    }
};
```

The approach is illustrated in Figure 21.13. Assume sizes have been normalized with respect to the box capacity. In Figure 21.13(a), the tournament tree depicted corresponds to six boxes, and five insertions of items of normalized sizes 0.6, 0.6, 0.55, 0.8, and 0.5 in that order. Now if an item v of size 0.45 is to be inserted, we first see that the root node a has a capacity at least 0.45, indicating it is possible to fit that item. We then check a's left child, node b. It has capacity 0.45, which means we can fit v in a box in the subtree rooted at b. Since node b's left child has capacity 0.4, we are forced to go right to c. We first examine c's left child, d, which corresponds to a box with capacity 0.45, meaning we can pack v in d. We then update the tournament tree as shown Figure 21.13(b).

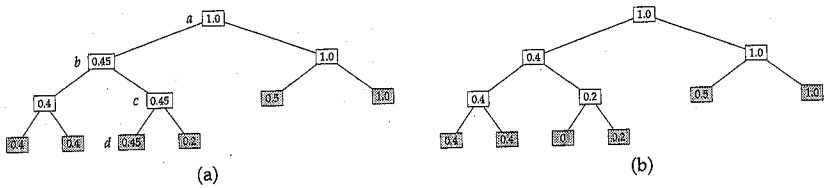


Figure 21.13: A tournament tree before and after inserting an item of size 0.45.

Problem 15.26, *pg. 127*: *Given a set of symbols with corresponding frequencies, find a code book that has the smallest average code length.*

Solution 15.26: Huffman coding yields an optimum solution to this problem. (There may be other optimum codes as well.) Huffman coding proceeds in three steps:

(1.) Sort characters in increasing order of frequencies and create a binary tree node for each character. Denote the set just created by S.

(2.) Create a new node n whose children are the two nodes with smallest frequencies and assign n's frequency to be the sum of the frequencies of its children.

(3.) Remove the children from S and add n to S. Repeat from Step (2.) till S consists of a single node, which is the root.

Mark all the left edges with 0 and the right edges with 1. The path from the root to a leaf node yields the bit string encoding the corresponding character.

We use a min-heap of candidate nodes to represent S. Since each invocation of Steps (2.) and (3.) requires two *extract-min* and one *insert* operation, we can find the Huffman codes in $O(n \log n)$ time. Here is an implementation of Huffman coding.

```
class Symbol {
  public:
    char c;
    double prob;
    string code;
};
```

```

class BinaryTree {
  public:
    double prob;
    shared_ptr<Symbol> s;
    BinaryTree *left, *right;
};

class Compare {
  public:
    const bool operator()(const BinaryTree* lhs,
                          const BinaryTree* rhs) const {
      return lhs->prob > rhs->prob;
    }
};

// Traverse tree and assign code
void assign_huffman_code(const BinaryTree* r, const string &s) {
  if (r) {
    // This node (i.e.,leaf) contains symbol
    if (r->s) {
      r->s->code = s;
    } else {  // non-leaf node
      assign_huffman_code(r->left, s + '0');
      assign_huffman_code(r->right, s + '1');
    }
  }
}

void Huffman_encoding(vector<Symbol> &symbols) {
  // Initially assign each symbol into min->heap
  priority_queue<BinaryTree*, vector<BinaryTree*>, Compare> min_heap;
  for (Symbol &s : symbols) {
    min_heap.emplace(new BinaryTree{s.prob, shared_ptr<Symbol>(&s),
                                    nullptr, nullptr});
  }

  // Keep combining two nodes until there is one node left
  while (min_heap.size() > 1) {
    BinaryTree* l = min_heap.top();
    min_heap.pop();
    BinaryTree* r = min_heap.top();
    min_heap.pop();
    min_heap.emplace(new BinaryTree{l->prob + r->prob, nullptr, l, r});
  }

  // Traverse the binary tree and assign code
  assign_huffman_code(min_heap.top(), string());
}
```

Applying this algorithm to the frequencies for English characters presented in Table 15.1 on Page 127 yields the Huffman tree in Figure 21.14 on the following page. The path from root to leaf yields that character's Huffman code, which is listed in Table 21.1 on the next page. For example, the codes for t, e, and z are 000, 100, and

001001000, respectively.

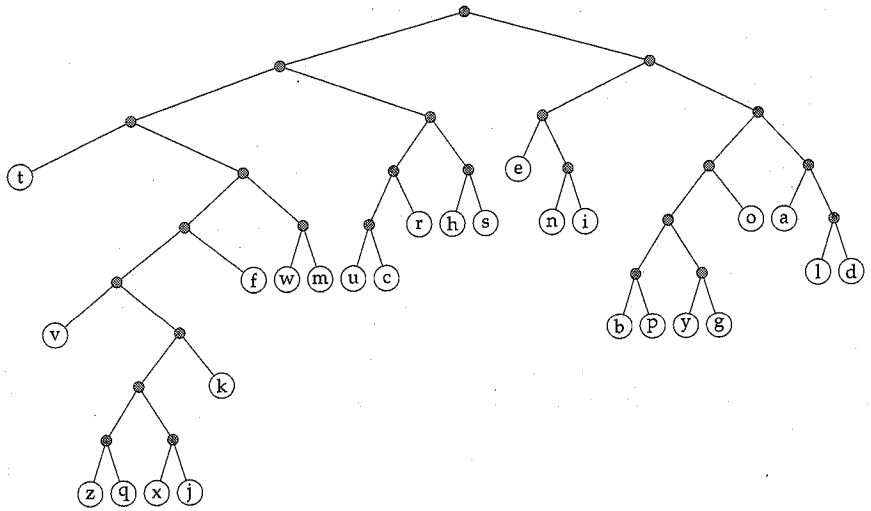


Figure 21.14: A Huffman tree for the English characters, assuming the frequencies given in Table 15.1 on Page 127.

Table 21.1: Huffman codes for English characters, assuming the frequencies given in Table 15.1 on Page 127.

Character	Huffman code	Character	Huffman code	Character	Huffman code
a	1110	j	001001011	s	0111
b	110000	k	0010011	t	000
c	01001	l	11110	u	01000
d	11111	m	00111	v	001000
e	100	n	1010	w	00110
f	00101	o	1101	x	001001010
g	110011	p	110001	y	110010
h	0110	q	001001001	z	001001000
i	1011	r	0101		

The codebook is explicitly given in Table 21.1. The average code length for this coding is 4.205. In contrast, the trivial coding takes $\lceil \log 26 \rceil = 5$ bits for each character.

Although it is unlikely that a rigorous proof of optimality would be asked in an interview setting, we give a proof by induction on the number of characters.

Proof:

For a single character, Huffman codes are trivially optimum. Let's say that for any distribution of frequencies among n characters, Huffman codes are optimum. Given this assumption, we will prove it is true for $n+1$ characters. We denote the frequency of character c by $f(c)$.

Suppose there exists an encoding that has a smaller average length of code for some frequency distribution for $n+1$ characters.

For any encoding, we can map the codes to a binary tree by identifying the

`null` string with root and adding a left edge for each 0 and a right edge for each 1.

We make several observations about a tree T corresponding to an optimum encoding:

- Each character must map to a leaf node; otherwise, the coding will violate our requirements on code prefixes.
- There cannot be a non-leaf node in T that has fewer than two children (otherwise, we can delete that node, bring its child one level up, and hence reduce the average code length).
- If we sort the leaves of T by their depths, the two deepest leaves must have the same depth (since the parent of the leaf with the longest path must have another child).
- The two deepest leaves in T must be assigned to two characters with smallest frequencies (otherwise, we can swap characters and achieve smaller average code length).
- Suppose we remove the two smallest frequency characters s and t and replace them with a new character u that has its frequency equal to $f(s) + f(t)$. Then the optimum prefix coding for this set C' of characters and corresponding frequencies must have the same average code length as the tree T' that results from deleting the two lowest frequency leaves in T (which as previously argued can be taken as siblings) and assigning their parent's frequency to be the sum of these two frequencies. Otherwise, if a tree S' for C' has lower average code length than T', we can create a tree S from S' by replacing the leaf corresponding to u with an internal node with two children corresponding to s and t. The average code length for the tree S is $f(s) + f(t)$ plus the average code length of the tree S', which, by hypothesis, has a lower average code length than T'. Since by construction, the average code length of T is $f(s) + f(t)$ plus the average code length of T', this contradicts the assumed optimality of T.

Now suppose the characters have frequencies $p_1 \geq p_2 \geq \cdots \geq p_{n+1}$. Let $A(p_1, \ldots, p_{n+1})$ be the optimum average code length for this frequency distribution and $\mathcal{H}(p_1, \ldots, p_{n+1})$ be the average code length for Huffman coding.

From the above observations, it follows that

$$A(p_1, \ldots, p_{n+1}) = A(p_1, \ldots, p_{n-1}, p_n + p_{n+1}) + (p_n + p_{n+1}).$$

From the construction of Huffman codes we know that

$$\mathcal{H}(p_1, \ldots, p_{n+1}) = \mathcal{H}(p_1, \ldots, p_{n-1}, p_n + p_{n+1}) + (p_n + p_{n+1}).$$

By our inductive assumption, $\mathcal{H}(p_1, \ldots, p_{n-1}, p_n + p_{n+1}) = A(p_1, \ldots, p_{n-1}, p_n + p_{n+1})$. Hence $\mathcal{H}(p_1, \ldots, p_{n+1}) = A(p_1, \ldots, p_{n+1})$. In other words, Huffman coding is optimum for $n + 1$ characters if it is optimum for n characters.

Problem 15.27, *pg. 127*: *How would you efficiently assign to each node u a new weight $w'(u)$ such that (1.) each root-to-leaf path has the same weight W^*, (2.) for all nodes u, $w'(u) \geq w(u)$, and (3.) $\sum_{u \in nodes(T)} w'(u)$ is minimum? See Figure 15.11 on Page 128 for an example.*

Solution 15.27: Let $\mu(u)$ be the weight of a u-to-leaf path with maximum weight under the weighing function w. Since $\mu(u) = w(u) + \max_{v \in children} \mu(v)$, we can compute μ for all vertices in the tree in a single pass in $O(n)$ time.

Observe that the root r has the largest μ-value. We will show how to construct new weights in which *every* root-to-leaf path has weight $\mu(r)$, while satisfying the constraints. Since we seek to minimize the total weight, we would never use a value greater than $\mu(r)$ for W^*.

We compute the new weights with a top-down traversal. We use a global variable s that records the updated weight of the path from the root up to, but not including the current node u. We assign the new weight of u, $w'(u)$, to $w(u) + \mu(r) - (s + \mu(u))$, update s, and recurse on the children. When we finish processing a node, we return to the parent, again updating the value of s.

Now we justify the algorithm. The term $\mu(r) - (s + \mu(u))$ is the "slack" at the node u. Basically, it tells us how much weight needs to be added to the heaviest u-to-leaf path to make its weight $\mu(r)$. Optimality cannot be lost in updating $w'(u)$ by the slack amount, since otherwise the slack will have to be added across u's subtrees, which will increase $\sum_{u \in nodes(T)} w'(u)$ if u has more than one child, and not change it if it has a single child.

ϵ-Variant 15.27.1: Compute a weighing function w' which minimizes $\sum_{u \in nodes(T)} w'(u)$ subject to the constraint that for each leaf l the weight of the path from root to l under w' equals the weight of the path from root to l under w.

Problem 15.28, *pg. 128*: *Devise an efficient algorithm that takes as input a set P of people and a set $F \subset P \times P$ of pairs of people and returns a largest subset of P within which each individual knows three or more other members of P and does not know three or more other members of P. The "knows" relation is not necessarily symmetric or transitive.*

Solution 15.28: We compute the optimum invitation list by iteratively removing people who cannot meet Leona's constraints until there is no one left to remove—the remaining set is the unique maximum set of people that Leona can invite.

Specifically, we iteratively remove anyone who knows fewer than three people in the current set and anyone who has fewer than three people they do not know in the current set. The process must converge since we start with a finite number of people and remove at least one person in each iteration. The remaining set satisfies Leona's constraints by construction.

It remains to show that the remaining set is maximum. We do this by proving that people who are removed could never be in a set that satisfies the constraints. We use induction on the k-th person removed.

Proof:

Let p_1 be the first person to be removed. Either p_1 knows fewer than three

people in the entire set or p_1 does not know fewer than three people in the entire set. Clearly p_1 cannot belong to any set that satisfies the constraints.

Inductively assume the first $i-1$ persons removed could not belong to any set that satisfies the constraints. Consider p_i, the i-th person removed. It must be that either fewer than three people know p_i in the current set or p_i does not know fewer than three people in the current set. But by induction, the current set includes any set satisfying Leona's constraints, so p_i cannot belong to a set satisfying Leona's constraints, and induction goes through.

Problem 15.29, *pg. 128*: *Let $G = (V, E)$ be an undirected graph. A two-coloring of G is a function assigning each vertex of G to* black *or* white. *Call a two-coloring* diverse *if each vertex has at least half its neighbors opposite in color to itself. Does every graph have a diverse coloring? How would you compute a diverse coloring, if it exists?*

Solution 15.29:

Here is a simple greedy algorithm for computing a diverse coloring. Start with an arbitrary two-coloring. If it is diverse, we are done. Define a vertex to be diverse under a coloring if at least half its neighbors are opposite in color to itself. Search for a non-diverse vertex v and flip v's color. Stop when the coloring is diverse.

It is implicit in the above description that every graph has a diverse coloring. Proving that the algorithm converges is slightly tricky, since a flip may increase the number of non-diverse vertices—see Figures 21.15(a) on the following page and 21.15(b) on the next page for an example. Hence we cannot prove that the greedy algorithm convergence by showing the number of non-diverse vertices keeps decreasing.

The key to proving convergence of the greedy algorithm is to focus on edges instead of vertices. For a given coloring, define an edge to be diverse if it connects vertices of different colors. We prove that the greedy approach does converge to a diverse coloring by counting the number of diverse edges.

Proof:

Suppose x is not diverse. Without loss of generality, suppose x is white. Then by changing x's color to black, the number of diverse edges strictly increases (since x had more white neighbors than black neighbors, and the diversity of other edges is unchanged).

Therefore the greedy algorithm must eventually converge, since the number of edges is finite. When it stops, all vertices are diverse.

This process is illustrated in Figure 21.15 on the following page. The coloring in Figure 21.15(a) on the next page is not diverse, since D is colored white, and has three white and two black neighbors; it is the only non-diverse vertex. Flipping D's color makes E non-diverse (Figure 21.15(b) on the following page). However, the number of non-diverse edges reduces from five to four. Flipping E's color reduces the number of non-diverse edges to three, and results in a diverse coloring (Figure 21.15(c) on the next page).

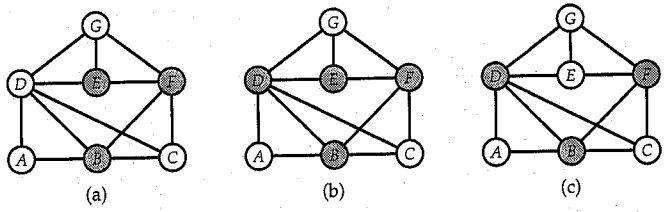


Figure 21.15: Computing a diverse coloring.

Problem 16.1, *pg. 132*: *Given a 2D array of black and white entries representing a maze with designated entrance and exit points, find a path from the entrance to the exit, if one exists.*

Solution 16.1: Model the maze as an undirected graph. Each vertex corresponds to a white pixel. We will index the vertices based on the coordinates of the corresponding pixel; so, vertex $v_{i,j}$ corresponds to the 2D array entry (i, j). Use edges to model adjacent pixels: $v_{i,j}$ is connected to vertices $v_{i+1,j}$, $v_{i,j+1}$, $v_{i-1,j}$, and $v_{i,j-1}$, assuming these vertices exist—vertex $v_{a,b}$ does not exist if the corresponding pixel is black or the coordinates (a, b) lie outside the image.

Now, run a DFS starting from the vertex corresponding to the entrance. If at some point, we discover the exit vertex in the DFS, then there exists a path from the entrance to the exit. If we implement recursive DFS then the path would consist of all the vertices in the call stack corresponding to previous recursive calls to the DFS routine.

This problem can also be solved using BFS from the entrance vertex on the same graph model. The BFS tree has the property that the computed path will be a shortest path from the entrance. However BFS is more difficult to implement than DFS since in DFS, the compiler implicitly handles the DFS stack, whereas in BFS, the queue has to be explicitly coded. Since the problem did not call for a shortest path, it is better to use DFS.

```
class Coordinate {
  public:
    int x, y;

    const bool operator==(const Coordinate& that) const {
      return (x == that.x && y == that.y);
    }
};

// Check cur is within maze and is a white pixel
bool is_feasible(const Coordinate &cur, const vector<vector<int>> &maze) {
  return cur.x >= 0 && cur.x < maze.size() &&
         cur.y >= 0 && cur.y < maze[cur.x].size() && maze[cur.x][cur.y] == 0;
}

// Perform DFS to find a feasible path
bool search_maze_helper(vector<vector<int>> &maze, const Coordinate &cur,
```

```
                                const Coordinate &e, vector<Coordinate> &path) {
  if (cur == e) {
    return true;
  }

  const array<array<int, 2>, 4> shift = {0, 1, 0, -1, 1, 0, -1, 0};
  for (const array<int, 2> &s : shift) {
    Coordinate next{cur.x + s[0], cur.y + s[1]};
    if (is_feasible(next, maze)) {
      maze[next.x][next.y] = 1;
      path.emplace_back(next);
      if (search_maze_helper(maze, next, e, path)) {
        return true;
      }
      path.pop_back();
    }
  }
  return false;
}

vector<Coordinate> search_maze(vector<vector<int>> maze, const Coordinate &s,
                               const Coordinate &e) {
  vector<Coordinate> path;
  maze[s.x][s.y] = 1;
  path.emplace_back(s);
  if (search_maze_helper(maze, s, e, path) == false) {
    path.pop_back();
  }
  return path;  // empty path means no path between s and e
}
```

Problem 16.2, *pg. 133*: *Given a dictionary D and two strings s and t, write a function to determine if s produces t. Assume that all characters are lowercase alphabets. If s does produce t, output the length of a shortest production sequence; otherwise, output* -1.

Solution 16.2: Define the undirected graph $G = (D, E)$ by $(u, v) \in E$ iff $|u| = |v|$, and u and v differ in one character. (Note that the relation "differs in one character" is symmetric, which is why the graph is undirected.)

A production sequence is simply a path in G, so what we need is a shortest path from s to t in G. Shortest paths in an undirected graph are naturally computed using BFS. We use a queue and a hash table of vertices (which indicates if a vertex has already been visited). We enumerate neighbors of a vertex v by an outer loop that iterates over each position in v and an inner loop that iterates over each choice of character for that position.

```
// Use BFS to find the least steps of transformation
int transform_string(unordered_set<string> D, const string &s,
                     const string &t) {
  queue<pair<string, int>> q;
  D.erase(s);  // mark s as visited by erasing it in D
  q.emplace(s, 0);

```

```
  while (!q.empty()) {
    pair<string, int> f(q.front());
    // Return if we find a match
    if (f.first == t) {
      return f.second;  // number of steps reaches t
    }

    // Try all possible transformations of f.first
    string str = f.first;
    for (int i = 0; i < str.size(); ++i) {
      for (int j = 0; j < 26; ++j) {  // iterates through 'a' ~ 'z'
        str[i] = 'a' + j;  // change the (i + 1)-th char of str
        auto it(D.find(str));
        if (it != D.end()) {
          D.erase(it);  // mark str as visited by erasing it
          q.emplace(str, f.second + 1);
        }
      }
      str[i] = f.first[i];  // revert the change of str
    }
    q.pop();
  }

  return -1;  // cannot find a possible transformations
}
```

Problem 16.3, *pg. 133*: *Design an algorithm that takes a set of pins and a set of wires connecting pairs of pins, and determines if it is possible to place some pins on the left half of a PCB, and the remainder on the right half, such that each wire is between left and right halves. Return such a division, if one exists. For example, the light vertices and dark vertices in Figure 16.5 on Page 133 are such division.*

Solution 16.3: Assuming the pins are numbered from 0 to $p-1$, create an undirected graph G on p vertices $v_0, \ldots, v_{p-1}$. Add an edge between v_i to v_j if Pin i and Pin j are connected by a wire. For simplicity, assume G is connected; if not, the connected components can be analyzed independently.

Run BFS on G starting with v_0. Assign v_0 arbitrarily to lie on the left half. All vertices at an odd distance from v_0 are assigned to the right half.

When performing BFS on an undirected graph, all newly discovered edges will either be from vertices which are at a distance d from v_0 to undiscovered vertices (which will then be at a distance $d+1$ from v_0) or from vertices which are at a distance d to vertices which are also at a distance d. First, assume we never encounter an edge from a distance k vertex to a distance k vertex. In this case, each wire is from a distance k vertex to a distance $k+1$ vertex, so all wires are between the left and right halves.

If any edge is from a distance k vertex to a distance k vertex, we stop—the pins cannot be partitioned into left and right halves as desired. The reason is as follows. Let u and v be such vertices. Consider the first common ancestor a in the BFS tree of u and v (such an ancestor must exist since the search started at v_0). The paths $p_{a,u}$ and

$p_{a,v}$ in the BFS tree from a to u and v are of equal length; therefore the cycle formed by going from a to u via $p_{a,u}$, then through the edge (u, v), and then back to a from v via $p_{a,v}$ has an odd length. A cycle in which the vertices can be partitioned into two sets must have an even number of edges—it has to go back and forth between the sets and terminate at the starting vertex, and each back and forth adds two edges. Therefore the vertices in an odd length cycle cannot be partitioned into two sets such that all edges are between the sets.

```
class GraphVertex {
  public:
    int d;
    vector<GraphVertex*> edges;

    GraphVertex(void) : d(-1) {}
};

bool BFS(GraphVertex* s) {
  queue<GraphVertex*> q;
  q.emplace(s);

  while (q.empty() == false) {
    for (GraphVertex* &t : q.front()->edges) {
      if (t->d == -1) {  // unvisited vertex
        t->d = q.front()->d + 1;
        q.emplace(t);
      } else if (t->d == q.front()->d) {
        return false;
      }
    }
    q.pop();
  }
  return true;
}

bool is_any_placement_feasible(vector<GraphVertex> &G) {
  for (GraphVertex &v : G) {
    if (v.d == -1) {  // unvisited vertex
      v.d = 0;
      if (BFS(&v) == false) {
        return false;
      }
    }
  }
  return true;
}
```

ϵ-Variant 16.3.1: Design an algorithm that checks if a graph is bipartite.

ϵ-Variant 16.3.2: Design an algorithm that checks if a graph is 2-colorable.

Problem 16.4, *pg. 133*: *Let $G = (V, E)$ be a connected undirected graph. How would you*

efficiently check if G is 2∃-connected? Can you make your algorithm run in $O(|V|)$ time? How would you check if G is 2∀-connected?

Solution 16.4: First, we consider the problem of checking if G is 2∃-connected. If $G' = (V, E \setminus \{(u,v)\})$ is connected, it must be that a path exists between u and v. This is possible iff u and v lie on a cycle in G. Thus we have reduced the problem of checking if G is 2∃-connected to the checking if there exists a cycle in G.

We can check for the existence of a cycle in G by running DFS on G. Recall DFS maintains a color for each vertex. Initially, all vertices are white. When a vertex is first discovered, it is colored gray. When DFS finishes processing a vertex, that vertex is colored black.

As soon as we discover an edge from a gray vertex back to a gray vertex which is not its immediate predecessor in the search, a cycle exists in G and we can stop.

In general, the time complexity of DFS is $O(|V| + |E|)$. However the algorithm described above has time complexity $O(|V|)$. This is because an undirected graph with no cycles can have at most $|V| - 1$ edges.

```
class GraphVertex {
  public:
    enum Color {white, gray, black} color;
    vector<GraphVertex*> edges;
};

bool DFS(GraphVertex* cur, const GraphVertex* pre) {
  // Visiting a gray vertex means a cycle
  if (cur->color == GraphVertex::gray) {
    return true;
  }

  cur->color = GraphVertex::gray;  // marks current vertex as a gray one
  // Traverse the neighbor vertices
  for (GraphVertex* &next : cur->edges) {
    if (next != pre && next->color != GraphVertex::black) {
      if (DFS(next, cur)) {
        return true;
      }
    }
  }
  cur->color = GraphVertex::black;  // marks current vertex as black
  return false;
}

bool is_graph_2_exist(vector<GraphVertex> &G) {
  if (G.empty() == false) {
    return DFS(&G.front(), nullptr);
  }
  return false;
}
```

Now, we consider the problem of checking if G is 2∀-connected. Clearly, G is not 2∀-connected iff there exists an edge e such that $G' = (V, E \setminus \{e\})$ is disconnected. The latter condition holds iff no cycle includes edge e.

We can find an edge (u, v) that is not on a cycle with DFS. Without loss of generality, assume u is discovered first. Observe that the removal of (u, v) disconnects G iff no back edges exist between v or v's descendants and u or u's ancestors. Define $l(v)$ to be the minimum of the discovery time $d(v)$ of v and $d(w)$ of w such that (t, w) is a back edge from t, where t is a descendant of v, and w is an ancestor of v.

We claim $l(v) < d(v)$ iff a back edge exists between v or one of v's descendants to u or one of u's ancestors. If $l(v) < d(v)$, then there exists a path from v through one of its descendants to an ancestor of v, i.e., v lies on a cycle. For every vertex v, except the root, if $l(v) = d(v)$, it is not possible to get from v back to u; hence removal of (u, v) disconnects u and v. If the root is connected to an edge whose removal disconnects G, the other vertex on that edge will be identified by the technique above, meaning we do not need to consider the root at all.

Now, we show how to compute $l(v)$ efficiently. Once we have processed all of v's children, then $l(v) = \min(d(v), \min_{x \in \text{children}(v)} l(x))$. This computation does not add to the asymptotic complexity of DFS since it is just constant additional work per edge, so we can check $2\forall$-connectedness in $O(|V| + |E|)$ time.

```
class GraphVertex {
  public:
    int d, l;  // discovery and leaving time
    vector<GraphVertex*> edges;
};

bool DFS(GraphVertex* cur, GraphVertex* pre, int time) {
  cur->d = ++time, cur->l = numeric_limits<int>::max();
  for (GraphVertex* &next : cur->edges) {
    if (next != pre) {
      if (next->d != 0) {  // back edge
        cur->l = min(cur->l, next->d);
      } else {  // forward edge
        if (DFS(next, cur, time) == false) {
          return false;
        }
        cur->l = min(cur->l, next->l);
      }
    }
  }
  return (pre == nullptr || cur->l < cur->d);
}

bool is_graph_2_for_all(vector<GraphVertex> &G) {
  if (G.empty() == false) {
    return DFS(&G.front(), nullptr, 0);
  }
  return true;
}
```

ϵ-Variant 16.4.1: Let G be a connected undirected graph. A vertex of G is an *articulation point* if its removal disconnects G. An edge of G is a *bridge* if its removal disconnects G. A *biconnected component* (BCC) of G is a maximal set of edges having

the property that any two edges in the set lie on common simple cycle. Design algorithms for computing articulation points, bridges, and BCCs.

Variant 16.4.2: Solve the 2∃-connectedness problem for an undirected graph using only two colors per vertex. (Do not use auxiliary data structures such as hash tables to mimic the third color.)

Problem 16.5, *pg. 134*: *Devise an efficient algorithm which takes a social network and computes for each individual his extended contacts.*

Solution 16.5: It is natural to model the network as a graph: vertices correspond to individuals, and an edge exists from A to B if B is a contact of A.

For an individual x, we can compute the set of x's contacts by running graph search (DFS or BFS) from x. Running graph search for each individual leads to a $O(|V|(|V| + |E|))$ algorithm for transitive closure.

```
class GraphVertex {
  public:
    int visitTime;
    vector<GraphVertex*> edges, extendedContacts;
};

void DFS(GraphVertex* cur, const int &time, vector<GraphVertex*> &contacts) {
  for (GraphVertex* &next : cur->edges) {
    if (next->visitTime != time) {
      next->visitTime = time;
      contacts.emplace_back(next);
      DFS(next, time, contacts);
    }
  }
}

void transitive_closure(vector<GraphVertex> &G) {
  // Build extended contacts for each vertex
  for (int i = 0; i < G.size(); ++i) {
    if (G[i].visitTime != i) {
      G[i].visitTime = i;
      DFS(&G[i], i, G[i].extendedContacts);
    }
  }
}
```

Another approach which has complexity $O(|V|^3)$ but which may be more efficient in practice for dense graphs, i.e., graphs in which $|E| = \Theta(|V|^2)$ is to run an all pairs shortest paths algorithm with edge weights of 1. If a path exists from u to v, the shortest path distance from u to v will be finite; otherwise, it will be ∞. We can improve this shortest path calculation by simply recording whether there is a path from u to v. In this way, we need a Boolean 2D array rather than an integer 2D array encoding the distances between the vertices. Although from an asymptotic perspective, the approach based on the all pairs shortest paths algorithm has the

same complexity as running multiple graph searches, it is likely to be more efficient in practice because of the use of Boolean arrays.

Problem 16.6, *pg. 134*: *Design an efficient algorithm that takes as input a collection of equality and inequality constraints and decides whether the constraints can be satisfied simultaneously.*

Solution 16.6: Let ϕ be a set of equality and inequality constraints on variables $x_0, \ldots, x_{n-1}$. Create an undirected graph G_ϕ on vertices $x_0, \ldots, x_{n-1}$; for each equality $x_i = x_j$, add the edge (x_i, x_j).

Now examine the connected components of G_ϕ. By the transitivity of equality, we can infer that $x_i = x_j$ for all vertices x_i and x_j in a common connected component.

Therefore if for some inequality $x_p \neq x_q$, vertices x_p and x_q lie in the same connected component, the set of constraints ϕ is not satisfied.

Conversely, let there be k connected components $C_0, \ldots, C_{k-1}$. Assign the variables in C_i to the value i. This satisfies all the equality constraints and since all the inequality constraints involve variables from different connected components, all inequality constraints are satisfied too.

```
class Constraint {
  public:
    int a, b;
};

class GraphVertex {
  public:
    int group;  // represents the connected component it belongs
    vector<GraphVertex*> edges;

    GraphVertex() : group(-1) {}
};

void DFS(GraphVertex &u) {
  for (GraphVertex* &v : u.edges) {
    if (v->group == -1) {
      v->group = u.group;
      DFS(*v);
    }
  }
}

bool are_constraints_satisfied(
  const vector<Constraint> &E,     // Equality constraints
  const vector<Constraint> &I) {  // Inequality constraints
  unordered_map<int, GraphVertex> G;
  // Build graph G according to E
  for (const Constraint &e : E) {
    G[e.a].edges.emplace_back(&G[e.b]), G[e.b].edges.emplace_back(&G[e.a]);
  }

  // Assign group index for each connected component
  int group_count = 0;
```

```
  for (pair<int, GraphVertex> vertex : G) {
    if (vertex.second.group == -1) {  // is a unvisited vertex
      vertex.second.group = group_count++;  // assigns a group index
      DFS(vertex.second);
    }
  }

  // Examine each inequality constraint to see if there is a violation
  for (const Constraint &i : I) {
    if (G[i.a].group == G[i.b].group) {
      return false;
    }
  }
  return true;
}
```

Problem 16.7, *pg. 135*: *How would you generalize your solution to Problem 13.6 on Page 100. to determine the largest number of teams that can be photographed simultaneously subject to the same constraints?*

Solution 16.7: Let G be the DAG with vertices corresponding to the teams as follows and edges from vertex X to Y iff $\texttt{sort}(X) < \texttt{sort}(Y)$.

Every sequence of teams where a team can be placed behind its predecessor corresponds to a path in G. To find the longest such sequence, we simply need to find the longest path in the DAG G. We can do this, for example, by topologically ordering the vertices in G; the longest path terminating at vertex v is the maximum of the longest paths terminating at v's fan-ins concatenated with v itself.

The topological ordering computation is $O(|V| + |E|)$ and dominates the computation time.

```
class GraphVertex {
  public:
    vector<GraphVertex*> edges;
    int maxDistance;
    bool visited;

    GraphVertex(void) : maxDistance(1), visited(false) {};
};

void DFS(GraphVertex* cur, stack<GraphVertex*> &vertex_order) {
  cur->visited = true;
  for (GraphVertex* &next : cur->edges) {
    if (next->visited == false) {
      DFS(next, vertex_order);
    }
  }
  vertex_order.emplace(cur);
}

stack<GraphVertex*> build_topological_ordering(vector<GraphVertex> &G) {
  stack<GraphVertex*> vertex_order;
  for (GraphVertex &g: G) {
```

```
    if (g.visited == false) {
      DFS(&g, vertex_order);
    }
  }
  return vertex_order;
}

int find_longest_path(stack<GraphVertex*> &vertex_order) {
  int max_distance = 0;
  while (vertex_order.empty() == false) {
    GraphVertex* u = vertex_order.top();
    max_distance = max(max_distance, u->maxDistance);
    for (GraphVertex* &v : u->edges) {
      v->maxDistance = max(v->maxDistance, u->maxDistance + 1);
    }
    vertex_order.pop();
  }
  return max_distance;
}

int find_largest_number_teams(vector<GraphVertex> &G) {
  stack<GraphVertex*> vertex_order(build_topological_ordering(G));
  return find_longest_path(vertex_order);
}
```

Problem 16.8, *pg. 136*: *Given an instance of the task scheduling problem, compute the least amount of time in which all the tasks can be performed, assuming an unlimited number of servers. Explicitly check that the system is feasible.*

Solution 16.8: This problem is naturally modeled using a directed graph. Vertices correspond to tasks, and an edge from u to v indicates that u must be completed before v can begin. The system is infeasible iff a cycle is present in the derived graph.

We can check the presence of a cycle by performing a DFS. If no cycle is present, the DFS numbering yields a topological ordering of the graph, i.e., an ordering of the vertices such that v follows u whenever an edge is present from u to v. Specifically, the DFS finishing time gives a topological ordering in reverse order. Therefore both testing for a cycle and computing a topological ordering can be performed in $O(n+m)$ time, where n and m are the number of vertices and edges in the graph, respectively.

Since the number of servers is unlimited, T_i can be completed τ_i time after all the tasks it depends on have completed. Therefore we can compute the soonest each task can complete by processing tasks in topological order, starting from the tasks that depend on no other tasks. If no such tasks exist, there must be a sequence of tasks starting and ending at the same task, such that each task requires the previous task to be completed before it can be started, i.e., the system is infeasible.

When the number of servers is limited, the problem becomes NP-complete. An equivalent problem with limited resources is the subject of Problem 17.13 on Page 143.

Problem 16.9, *pg. 136*: *Design an algorithm which takes as input a graph $G = (V, E)$, directed or undirected, a nonnegative cost function on E, and vertices s and t; your algorithm*

should output a path with the fewest edges amongst all shortest paths from s to t.

Solution 16.9: Dijkstra's shortest path algorithm uses scalar values for edge length. However it can easily be modified to the case where the edge weight is a pair if *addition* and *comparison* can be defined over these pairs. In this case, if the edge cost is c, we say the length of the edge is given by the pair $(c, 1)$. We define addition to be just component-wise addition. Hence if we sum up the edge lengths over a path, we essentially get the total cost and the number of edges in the path. The compare function is lexicographic, first the total cost, then the number of edges. We can run Dijkstra's shortest path algorithm with this compare function and find the shortest path that requires the least number of edges.

Since a heap does not support efficient updates, it is more convenient to use a BST than a heap to implement the algorithm.

```
template <typename DistanceType>
class GraphVertex {
  public:
    pair<DistanceType, int> distance;  // stores (dis, #edges) pair
    // stores (vertex, dis) pair
    vector<pair<GraphVertex<DistanceType>*, DistanceType>> edges;
    int id;  // stores the id of this vertex
    GraphVertex* pred;  // stores the predecessor in the shortest path
    bool visited;

    GraphVertex(void) :
      distance(numeric_limits<DistanceType>::max(), 0),
      pred(nullptr),
      visited(false) {}
};

template <typename DistanceType>
class Comp {
  public:
    const bool operator()(const GraphVertex<DistanceType>* lhs,
                          const GraphVertex<DistanceType>* rhs) const {
      return lhs->distance.first < rhs->distance.first ||
             (lhs->distance.first == rhs->distance.first &&
                 lhs->distance.second < rhs->distance.second);
    }
};

template <typename DistanceType>
void output_shortest_path(GraphVertex<DistanceType>* &v) {
  if (v) {
    output_shortest_path(v->pred);
    cout << v->id << " ";
  }
}

template <typename DistanceType>
void Dijkstra_shortest_path(vector<GraphVertex<DistanceType>> &G,
                            GraphVertex<DistanceType>* s,
                            GraphVertex<DistanceType>* t) {
```

```
  // Initialization the distance of starting point
  s->distance = {0, 0};
  set<GraphVertex<DistanceType>*, Comp<DistanceType>> node_set;
  node_set.emplace(s);

  do {
    GraphVertex<DistanceType>* u = nullptr;
    // Extract the minimum distance vertex from heap
    while (node_set.empty() == false) {
      u = *node_set.cbegin();
      node_set.erase(node_set.cbegin());
      if (u->visited == false) {  // found an unvisited node
        break;
      }
    }

    if (u) {  // u is a valid vertex
      u->visited = true;  // mark u as visited
      // Relax neighboring vertices of u
      for (const auto &v : u->edges) {
        DistanceType v_distance = u->distance.first + v.second;
        int v_num_edges = u->distance.second + 1;
        if (v.first->distance.first > v_distance ||
            (v.first->distance.first == v_distance &&
             v.first->distance.second > v_num_edges)) {
          node_set.erase(v.first);
          v.first->pred = u;
          v.first->distance = {v_distance, v_num_edges};
          node_set.emplace(v.first);
        }
      }
    } else {  // u is not a valid vertex
      break;
    }
  } while (t->visited == false);  // until t is visited

  // Output the shortest path with fewest edges
  output_shortest_path(t);
}
```

ϵ-Variant 16.9.1: Solve the same problem when edge weights are integers in $(-\infty, \infty)$. You may modify the graph, but must use an unmodified shortest path algorithm.

Problem 16.10, *pg. 136*: *Given a time-table, a starting city, a starting time, and a destination city, how would you compute the soonest you could get to the destination city? Assume all flights start and end on time, and that you need 60 minutes between flights.*

Solution 16.10: We use Dijkstra's single-source shortest path algorithm with a minor variation. At each iteration we add a new city c to the set of cities to which we know the fastest route. After c is identified, we relax edges out of c by searching for all flights out of c that depart 60 minutes or later from the earliest we can get to c. We do

not need to relax an edge from c to c' more than once, e.g., if two flights from c to c' that satisfy the inter-flight time constraint, we only need to relax the one that arrives in c' sooner.

The time complexity is identical to that of Dijkstra's algorithm with the flights playing the role of edges, and the cities playing the role of vertices.

Problem 16.11, *pg. 136*: *Devise an efficient algorithm which takes the existing highway network (specified as a set of highway sections between pairs of cities) and proposals for new highway sections, and returns a proposed highway section which minimizes the shortest driving distance between El Paso and Corpus Christi.*

Solution 16.11: Note that we cannot add more than one proposal to the existing network and run a shortest path algorithm—we may end up with a shortest path which uses multiple proposals.

Let there be n cities, m existing highway sections, and k proposed sections. One approach is to simply run a single-source shortest path algorithm from El Paso k times, one for each of the proposed sections. Dijkstra's single-source shortest path algorithm can be made to run in $O(n \log n + m)$, leading to an overall time complexity of $O(k(n \log n + m))$.

Since $m+k$ can be as large as $\frac{n(n-1)}{2}$, the above approach can have a time complexity as high as $O(n^4)$. We can improve upon this by first running an all pairs shortest paths algorithm, such as the Floyd-Warshall algorithm which will compute all pairs of shortest paths distances in $O(n^3)$ time.

Let $S(u, v)$ be the 2D array of shortest path distances for each pair of cities. Each proposal p is a pair of cities (x, y). The best we can do by using proposal p is $\min(S(a, b), S(a, x) + d(x, y) + S(y, b))$ where $d(x, y)$ is the distance of the proposed highway p between x and y, and a and b are El Paso and Corpus Christi, respectively. This computation is $O(1)$ time, so we can evaluate all the proposals in time proportional to the number of proposals after we have computed the shortest path between each pair of cities. This results in an $O(n^3 + k)$ time complexity; since $k \leq \frac{n(n-1)}{2}$, the time complexity is $O(n^3)$.

```
template <typename DistanceType>
class HighwaySection {
  public:
    int x, y;
    DistanceType distance;
};

template <typename DistanceType>
void Floyd_Warshall(vector<vector<DistanceType>> &G) {
  for (int k = 0; k < G.size(); ++k) {
    for (int i = 0; i < G.size(); ++i) {
      for (int j = 0; j < G.size(); ++j) {
        if (G[i][k] != numeric_limits<DistanceType>::max() &&
            G[k][j] != numeric_limits<DistanceType>::max() &&
            G[i][j] < G[i][k] + G[k][j]) {
          G[i][j] = G[i][k] + G[k][j];
        }
```

```
      }
    }
  }
}

template <typename DistanceType>
HighwaySection<DistanceType> find_best_proposals(
    const vector<HighwaySection<DistanceType>> &H,
    const vector<HighwaySection<DistanceType>> &P,
    const int &a, const int &b, const int &n) {
  // G stores the shortest path distance between all pairs
  vector<vector<DistanceType>>
    G(n, vector<DistanceType>(n, numeric_limits<DistanceType>::max()));

  // Build graph G based on existing highway sections H
  for (const HighwaySection<DistanceType> &h : H) {
    G[h.x][h.y] = G[h.y][h.x] = h.distance;
  }
  // Perform Floyd Warshall to build the shortest path between vertices
  Floyd_Warshall(G);

  // Examine each proposal for shorter distance between a and b
  DistanceType min_dis_a_b = G[a][b];
  HighwaySection<DistanceType> best_proposal;
  for (const HighwaySection<DistanceType> &p : P) {
    if (G[a][p.x] != numeric_limits<DistanceType>::max() &&
        G[p.y][b] != numeric_limits<DistanceType>::max()) {
      if (min_dis_a_b < G[a][p.x] + p.distance + G[p.y][b]) {
        min_dis_a_b = G[a][p.x] + p.distance + G[p.y][b];
        best_proposal = p;
      }
      if (min_dis_a_b < G[a][p.y] + p.distance + G[p.x][b]) {
        min_dis_a_b = G[a][p.y] + p.distance + G[p.x][b];
        best_proposal = p;
      }
    }
  }
  return best_proposal;
}
```

Problem 16.12, *pg. 137*: *Design an efficient algorithm to determine whether there exists an arbitrage—a way to start with a single unit of some commodity C and convert it back to more than one unit of C through a sequence of exchanges.*

Solution 16.12: We define a weighted directed graph $G = (V, E = V \times V)$, where V corresponds to the set of commodities. The weight $w(e)$ of edge $e = (u, v)$ is the amount of commodity v we can buy with one unit of commodity u. Observe that an arbitrage exists iff there exists a cycle in G whose edge weights multiply out to more than 1.

Create a new graph $G' = (V, E)$ with weight function $w'(e) = -\lg w(e)$. Since $\lg(a \times b) = \lg a + \lg b$, there exists a cycle in G whose edge weights multiply out to more than 1 iff there exists a cycle in G' whose edge weights sum up to less than

$\lg 1 = 0$. (This property is true for logarithms to any base, so if it is more efficient for example to use base-e, we can do so.)

The Bellman-Ford algorithm, which takes $O(|V||E|)$ time, detects negative-weight cycles. Usually finding a negative-weight cycle is done by adding a dummy vertex s with 0-weight edges to each vertex in the given graph and running the Bellman-Ford single-source shortest path algorithm from s. However, for the arbitrage problem, the graph is complete. Hence we can run Bellman-Ford algorithm from any single vertex, and get the right result.

```
bool Bellman_Ford(const vector<vector<double>> &G, const int &source) {
  vector<double> dis_to_source(G.size(), numeric_limits<double>::max());
  dis_to_source[source] = 0;

  for (int times = 1; times < G.size(); ++times) {
    bool have_update = false;
    for (int i = 0; i < G.size(); ++i) {
      for (int j = 0; j < G[i].size(); ++j) {
        if (dis_to_source[i] != numeric_limits<double>::max() &&
            dis_to_source[j] > dis_to_source[i] + G[i][j]) {
          have_update = true;
          dis_to_source[j] = dis_to_source[i] + G[i][j];
        }
      }
    }

    // No update in this iteration means no negative cycle
    if (have_update == false) {
      return false;
    }
  }

  // Detect cycle if there is any further update
  for (int i = 0; i < G.size(); ++i) {
    for (int j = 0; j < G[i].size(); ++j) {
      if (dis_to_source[i] != numeric_limits<double>::max() &&
          dis_to_source[j] > dis_to_source[i] + G[i][j]) {
        return true;
      }
    }
  }
  return false;
}

bool is_Arbitrage_exist(vector<vector<double>> G) {
  // Transform each edge in G
  for (vector<double> &edge_list : G) {
    for (double &edge : edge_list) {
      edge = -log10(edge);
    }
  }

  // Use Bellman-Ford to find negative weight cycle
  return Bellman_Ford(G, 0);
```

```
}
```

Problem 16.13, *pg. 137*: *Let $G = (V, E)$ be an undirected graph with edge weight function $w : E \mapsto \mathcal{Z}^+$. You are given $T \subset E$, an MST of G. Let e be an edge. Design efficient algorithms for computing the MST when (1.) $w(e)$ decreases , and (2.) $w(e)$ increases.*

Solution 16.13: We make use of two key facts about an MST. The first is that if an edge e is the unique heaviest weight edge on some cycle, it cannot lie in the MST. The second is that if an edge is the unique lightest weight edge in some set of edges that disconnect the graph, it must lie in the MST.

For the first case, if $e \in T$, then T remains unchanged. Otherwise, let $e = (u, v)$. We search for the unique path π in T between u and v. If $w(e)$ remains greater than or equal to the weights of the edges on this path, T is unchanged. Otherwise, we obtain the new MST by deleting the edge in π which has maximum weight, and adding e.

For the second case, if $e \notin T$, then T will not change. Otherwise, let $e = (u, v)$. Removing e from T leaves us with two components. We want to replace e by the lowest-weight edge between these components. We can find this edge by first finding the two components, e.g., by DFS through $T \setminus \{e\}$ from u and from v, and then enumerating all the edge in G.

In both cases, the time complexity is dominated by the need to do graph search, i.e., $O(|V| + |E|)$.

Problem 17.1, *pg. 139*: *How would you programmatically determine if a tie is possible in a presidential election with two candidates, R and D?*

Solution 17.1: We need to determine if there exists a subset of states whose Electoral College votes add up to $\frac{538}{2} = 269$. This is an instance of the subset sum problem, and is known to be NP-complete. It is a specialization of the 0-1 knapsack problem described in Problem 17.2 on Page 139 and the DP solution to that problem can be used. Following is the code in C++:

```
// V contains the number of votes for each state
long ties_election(const vector<int> &V) {
  int total_votes = accumulate(V.cbegin(), V.cend(), 0);

  // No way to tie if the total number of votes is odd
  if (total_votes & 1) {
    return 0;
  }

  vector<vector<long>> table(V.size() + 1, vector<long>(total_votes + 1, 0));
  table[0][0] = 1;  // base condition: 1 way to reach 0
  for (int i = 0; i < V.size(); ++i) {
    for (int j = 0; j <= total_votes; ++j) {
      table[i + 1][j] = table[i][j] + (j >= V[i] ? table[i][j - V[i]] : 0);
    }
  }
  return table[V.size()][total_votes >> 1];
}
```

Problem 17.2, *pg. 139*: *Design an algorithm for the knapsack problem that selects a subset of items that has maximum value and weighs at most w ounces. All items have integer weights and values.*

Solution 17.2: Let $V[i, w]$ be the maximum value that can be packed with weight less than or equal to w using the first i clocks. Then $V[i, w]$ satisfies the following recurrence:

$$V[i,w] = \begin{cases} \max(V[i-1,w], V[i-1,w-w_i]+v_i), & \text{if } w_i \le w; \\ V[i-1,w], & \text{otherwise.} \end{cases}$$

For $i = 0$ or $w = 0$, we set $V[i, w] = 0$. This DP procedure computes $V[n, w]$ in $O(nw)$ time, and uses $O(nw)$ space. Note that the space complexity can be improved to $O(w)$ by using a one-dimensional array to store the current optimal result and rewriting the next step result back to this array. Following is the code in C++:

```
template <typename ValueType>
ValueType knapsack(const int &w, const vector<pair<int, ValueType>> &items) {
  vector<ValueType> V(w + 1, 0);
  for (int i = 0; i < items.size(); ++i) {
    for (int j = w; j >= items[i].first; --j) {
      V[j] = max(V[j], V[j - items[i].first] + items[i].second);
    }
  }
  return V[w];
}
```

Variant 17.2.1: Solve the knapsack problem when the thief can take a fractional amount of an item.

Problem 17.3, *pg. 140*: *Let array A be an array of n positive integers. Entry $A[i]$ is the value of the i-th stolen item. Design an algorithm that computes a subset $S \subset \mathcal{Z}_n = \{0, 1, 2, \ldots, n-1\}$ such that $|\sum_{i \in S} A[i] - \sum_{j \in \mathcal{Z}_n \setminus S} A[j]|$ is minimized.*

Solution 17.3: We first compute a Boolean-valued array *is_Ok*, indexed from 0 to sum $= \sum_{i=0}^{n-1} A[i]$, inclusive. The array *is_Ok* encodes whether a given i is the sum of the elements in some subset of stolen items. The *is_Ok* array entries are initialized to `false` for each index greater than 0; *is_Ok*[0] is initialized to `true`. We assign values to *is_Ok* as follows. Set *is_Ok*[$A[0]$] to `true`. Then set *is_Ok*[$A[1]$] and *is_Ok*[$A[0] + A[1]$] to `true`. Then set *is_Ok*[$A[2]$], *is_Ok*[$A[0] + A[2]$], *is_Ok*[$A[1] + A[2]$], and *is_Ok*[$A[0] + A[1] + A[2]$] to `true`. Generalizing, the set of values corresponding to subsets of $\{0, 1, \ldots, i-1, i\}$ is the union of the set of values $\{v_0, v_1, \ldots, v_{j-1}\}$ corresponding to subsets of $\{0, 1, \ldots, i-1\}$ and the set $\{v_0 + A[i], v_1 + A[i], \ldots, v_{j-1} + A[i]\}$. This new set of values can be computed by iterating over values v in $[\text{sum}, A[i]]$, setting *is_Ok*[v] to `true` wherever *is_Ok*[$v - A[i]$] is `true`. The time complexity is $O(n \cdot \text{sum})$.

After obtaining *is_Ok* we find the best partition by iterating over i from $\frac{\text{sum}}{2}$ downwards until we first hit an entry i such that *is_Ok*[i] is `true`; this is the closest we can get to an equal split. Note that *is_Ok* is symmetric about its midpoint since if $S \subset \{0,1,2,\ldots,n-1\}$ has a value w, then $S' = \mathcal{Z}_n \setminus S$ has a value $\text{sum} - w$, so we do not need to search *is_Ok* for indices greater than $\frac{\text{sum}}{2}$.

```
int minimize_difference(const vector<int> &A) {
  int sum = accumulate(A.cbegin(), A.cend(), 0);

  unordered_set<int> is_Ok;
  is_Ok.emplace(0);
  for (const int &item : A) {
    for (int v = sum >> 1; v >= item; --v) {
      if (is_Ok.find(v - item) != is_Ok.cend()) {
        is_Ok.emplace(v);
      }
    }
  }

  // Find the first i from middle where is_Ok[i] == true
  for (int i = sum >> 1; i > 0; --i) {
    if (is_Ok.find(i) != is_Ok.cend()) {
      return (sum - i) - i;
    }
  }
  return sum;   // one thief takes all
}
```

Variant 17.3.1: Solve the same problem with the additional constraint that the thieves have the same number of items.

Problem 17.4, *pg. 140*: *Write a program that determines a sequence of steps by which the required amount of milk can be obtained using the worn-out jugs. The milk is being added to a large mixing bowl, and hence cannot be removed from the bowl. Furthermore, it is not possible to pour one jug's contents into another. Your scheme should always work, i.e., return between* 2100 *and* 2300 *mL of milk, independent of how much is chosen in each individual step, as long as that quantity satisfies the given constraints.*

Solution 17.4: It is natural to solve this problem using recursion—if we use jug A for the last step, we need to correctly measure a volume of milk that is at least $2100-230 = 1870$ mL—the last measurement may be as little as 230 mL, and anything less than 1870 mL runs the risk of being too little. Similarly, the volume must be at most $2300 - 240 = 2060$ mL. The volume is not achievable if it is not achievable with any of the three jugs as ending points. We cache intermediate computations to reduce the number of recursive calls.

In the following code, we implement a general purpose function which finds the feasibility among n jugs; those arrays are passed in as `jugs`.

```
class Jug {
```

```
  public:
    int low, high;
};

class PairEqual {
  public:
    const bool operator()(const pair<int, int> &a,
                          const pair<int, int> &b) const {
      return a.first == b.first && a.second == b.second;
    }
};

class HashPair {
  public:
    const size_t operator()(const pair<int, int> &p) const {
      return hash<int>()(p.first) ^ hash<int>()(p.second);
    }
};

bool check_feasible_helper(const vector<Jug> &jugs, const int &L,
                           const int &H, unordered_set<pair<int, int>,
                                                       HashPair,
                                                       PairEqual> &c) {
  if (L > H || c.find({L, H}) != c.cend() || (L < 0 && H < 0)) {
    return false;
  }

  // Check the volume for each jug to see if it is possible
  for (const Jug &j : jugs) {
    if ((L <= j.low && j.high <= H) ||  // base case: j is contained in [L, H]
      check_feasible_helper(jugs, L - j.low, H - j.high, c)) {
      return true;
    }
  }
  c.emplace(L, H);  // marks this as impossible
  return false;
}

bool check_feasible(const vector<Jug> &jugs, const int &L, const int &H) {
  unordered_set<pair<int, int>, HashPair, PairEqual> cache;
  return check_feasible_helper(jugs, L, H, cache);
}
```

Variant 17.4.1: Suppose Jug i can be used to measure any quantity in $[l_i, u_i]$ exactly. Determine if it is possible to measure a quantity of milk between L and U.

Problem 17.5, *pg. 141*: *Given a graph $G = (V, E)$, with cost function $c : E \mapsto \mathcal{Z}^+$, delay function $d : E \mapsto \mathcal{Z}^+$, designated vertices s and t, and a delay constraint $\Delta \in \mathcal{Z}^+$, find a path from s to t with minimum cost, subject to the constraint that the delay of the path is no more than Δ. Costs are additive—the cost of a path is the sum of the costs of the individual edges; the same holds for delays.*

Solution 17.5: The delay-constrained shortest-path problem is known to be NP-complete. It can be solved in pseudo-polynomial time using DP. For each vertex v and delay $\delta \in [0, \Delta]$ we compute $M(v, \delta)$, the minimum cost path from s to v that has a delay less than or equal to δ. The function $M(v, \delta)$ satisfies the following recurrence:

$$M(v, \delta) = \min_{u \in \text{fan-in}(v)} (c(u, v) + M(u, \delta - d(u, v)))$$

Base cases include $M(s, \delta) = 0$ and $M(v, \delta) = \infty$, for $\delta < 0$.

By caching values, $M(t, \Delta)$ can be computed in $O(|E|\Delta)$ time with $O(|V|\Delta)$ space for the cache.

Alternately, we can use branch and bound. We can obtain good lower bounds using Lagrangian relaxation. The general idea behind Lagrangian relaxation is to solve constrained optimization problems by adding the constraint as a penalty term in the objective function. In our specific setting we can use a multiplier $\lambda \in \mathcal{R}^+$ to compute a new cost $c(u, v) + \lambda d(u, v)$ for each edge. Now we run a conventional shortest path algorithm. The effect of the λ term is to avoid paths that incur a large delay—the larger λ is, the more we avoid delay.

Let c_{opt} be the optimum cost for delay feasible paths, and d_{opt} the delay on an optimum path. Suppose the shortest path for a given λ has cost $c^* + \lambda d^*$. Observe that if this path is infeasible, i.e., $d^* > \Delta$, then $c_{\text{opt}} > c^*$, since otherwise it would be impossible for $c_{\text{opt}} + \lambda d_{\text{opt}} \geq c^* + \lambda d^*$.

Note that each $c^* + \lambda d^*$ where $d^* > \Delta$ yields a lower bound. The greatest lower bound across all λ may not be c_{opt}. This phenomenon is known as a duality gap. Furthermore, if $d^* > \Delta$, then we can say nothing about the relationship between c^* and c_{opt}.

Problem 17.6, *pg. 141*: *Suppose you are given a set of cities in the Cartesian plane, as shown in Figure 4.3 on Page 36. The cost of traveling from one city to another is a constant multiple of the distance between the cities. Give an efficient procedure for computing a tour whose cost is no more than two times the cost of an optimum tour.*

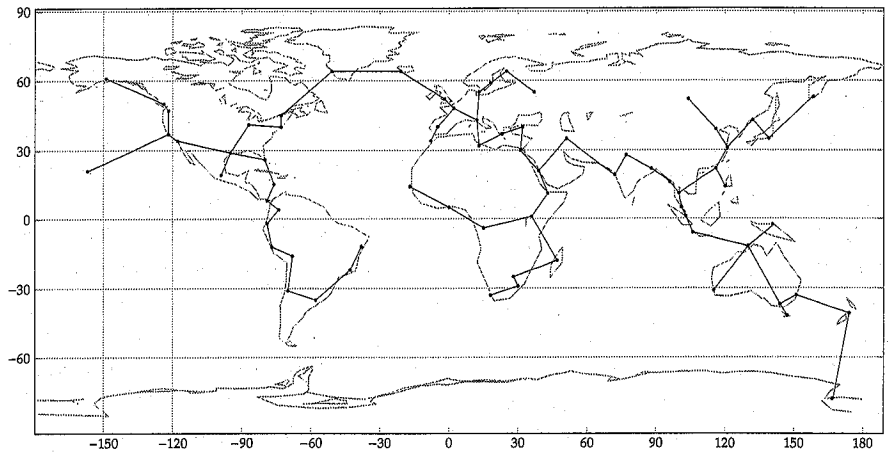
Solution 17.6: A good way to approach this problem is to think of a similar problem that can be solved exactly efficiently. The MST problem has an efficient algorithm, and it yields a way of visiting each city exactly twice—start at any city c and do an in-order walk in the MST with c as the root. This traversal leads to a path in which each edge is visited exactly twice.

Consider any tour. If we drop the edge back to the starting city, the remaining set of edges constitute a tree. Therefore the cost of the optimum tour is at least as great as the cost of the MST.

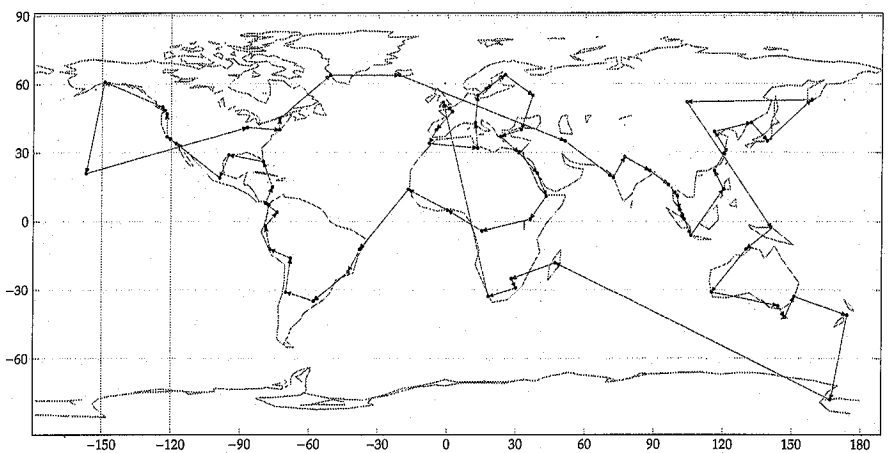
Now we make use of the fact that the distances between cities satisfies the triangle inequality to build a tour from the MST whose cost is no greater than the MST. When we perform our in-order walk, we simply skip over cities we have already visited—the direct distance from u to v cannot be more than the sum of distances on a path from u to v.

Hence we have a true tour costing at most twice the cost of the MST which itself was a lower bound on the cost of the traveling salesman problem, i.e., the tour has a cost that is at most twice the cost of an optimum tour.

The application of the algorithm described above to the cities in Figure 4.3 on Page 36 is shown in Figure 21.16.



(a) A minimum spanning tree for the cities in Figure 4.3 on Page 36.



(b) A tour derived from the minimum spanning tree in (a).

Figure 21.16: The two key steps of the approximation algorithm for the TSP.

Problem 17.7, *pg. 142*: *Design a fast algorithm for selecting k warehouse locations that is provably within a constant factor of the optimum solution.*

Solution 17.7: A natural approach to this problem is to build the assignment one warehouse at a time. We can pick the first warehouse to be the city for which the cost

is minimized—this takes $\Theta(n^2)$ time since we try each city one at a time and check its distance to every other city.

Let's say we have selected the first $i-1$ warehouses $\{w_1, w_2, \ldots, w_{i-1}\}$ and are trying to choose the i-th warehouse. A reasonable choice for the i-th warehouse is the one that is the farthest from the $i-1$ warehouses already chosen. This city can be computed in $O(ni)$ time.

We use the computation above to select k warehouses. Let the maximum distance from the remaining cities to the k warehouses be d_m. Then the cost of this assignment is d_m. Let e be a city which is d_m distance from the warehouse it is closest to. Note that the k warehouse cities are all at least d_m apart; otherwise, we would have chosen e to locate a warehouse at in one of the first k iterations.

By the pigeonhole principle, at least two of the $k+1$ cities $\{w_1, w_2, \ldots, w_k, e\}$ must have the same closest warehouse in an optimum assignment. Let p and q be two such cities and w be the warehouse city closest to p and q in an optimum assignment of warehouses. By the triangle inequality, $d(p,q) \leq d(w,p) + d(w,q)$. Since $d_m \leq d(p,q)$, it follows that at least one of $d(w,p)$ or $d(w,q)$ is greater than or equal to $\frac{d_m}{2}$. Hence the cost of this optimum assignment is at least $\frac{d_m}{2}$, implying our greedy heuristic produced an assignment that is within a factor of two of the cost of the optimum assignment.

Note that the initial selection of a warehouse is immaterial for the argument to work but heuristically, it is better to choose a central city as a starting point.

Problem 17.8, *pg. 142*: *Implement a Sudoku solver. Your program should read an instance of Sudoku from the command line. The command line argument is a sequence of 3-digit strings, each encoding a row, a column, and a digit at that location.*

Solution 17.8: We use a straight-forward application of the backtracking principle. We traverse the 2D array entries one at a time. If the entry is empty, we try each value for the entry, and see if the updated 2D array is still valid; if it is we recurse. If all the entries have been filled, the search is successful.

In practice it is more efficient to see if a conflict results on adding a new entry before adding it rather than adding it and seeing if a conflict is present. See the code for details.

```
bool valid_to_add(const vector<vector<int>> &A, const int &i, const int &j,
                  const int &val) {
  // Check row constraints
  for (int k = 0; k < A.size(); ++k) {
    if (val == A[k][j]) {
      return false;
    }
  }

  // Check column constraints
  for (int k = 0; k < A.size(); ++k) {
    if (val == A[i][k]) {
      return false;
    }
```

```
  }

  // Check region constraints
  int region_size = sqrt(A.size());
  int I = i / region_size, J = j / region_size;
  for (int a = 0; a < region_size; ++a) {
    for (int b = 0; b < region_size; ++b) {
      if (val == A[region_size * I + a][region_size * J + b]) {
        return false;
      }
    }
  }
  return true;
}

bool solve_Sudoku_helper(vector<vector<int>> &A, int i, int j) {
  if (i == A.size()) {
    i = 0;  // start a new row
    if (++j == A[i].size()) {
      return true;  // entire matrix has been filled without conflict
    }
  }

  // Skip nonempty entries
  if (A[i][j] != 0) {
    return solve_Sudoku_helper(A, i + 1, j);
  }

  for (int val = 1; val <= A.size(); ++val) {
    // Note: practically, it's substantially quicker to check if entryval
    // conflicts with any of the constraints if we add it at (i,j) before
    // adding it, rather than adding it and then calling is_valid_Sudoku.
    // The reason is that we know we are starting with a valid configuration,
    // and the only entry which can cause a problem is entryval at (i,j).
    if (valid_to_add(A, i, j, val)) {
      A[i][j] = val;
      if (solve_Sudoku_helper(A, i + 1, j)) {
        return true;
      }
    }
  }

  A[i][j] = 0;  // undo assignment
  return false;
}

// Check if a partially filled matrix has any conflicts
bool is_valid_Sudoku(const vector<vector<int>> &A) {
  // Check row constraints
  for (int i = 0; i < A.size(); ++i) {
    vector<bool> is_present(A.size() + 1, false);
    for (int j = 0; j < A.size(); ++j) {
      if (A[i][j] != 0 && is_present[A[i][j]] == true) {
        return false;
      } else {
```

```
        is_present[A[i][j]] = true;
      }
    }
  }

  // Check column constraints
  for (int j = 0; j < A.size(); ++j) {
    vector<bool> is_present(A.size() + 1, false);
    for (int i = 0; i < A.size(); ++i) {
      if (A[i][j] != 0 && is_present[A[i][j]] == true) {
        return false;
      } else {
        is_present[A[i][j]] = true;
      }
    }
  }

  // Check region constraints
  int region_size = sqrt(A.size());
  for (int I = 0; I < region_size; ++I) {
    for (int J = 0; J < region_size; ++J) {
      vector<bool> is_present(A.size() + 1, false);
      for (int i = 0; i < region_size; ++i) {
        for (int j = 0; j < region_size; ++j) {
          if (A[region_size * I + i][region_size * J + j] != 0 &&
              is_present[A[region_size * I + i][region_size * J + j]]) {
            return false;
          } else {
            is_present[A[region_size * I + i][region_size * J + j]] = true;
          }
        }
      }
    }
  }
  return true;
}

bool solve_Sudoku(vector<vector<int>> &A) {
  if (is_valid_Sudoku(A) == false) {
    cout << "Initial configuration violates constraints." << endl;
    return false;
  }

  if (solve_Sudoku_helper(A, 0, 0)) {
    for (int i = 0; i < A.size(); ++i) {
      copy(A[i].begin(), A[i].end(), ostream_iterator<int>(cout, " "));
      cout << endl;
    }
    return true;
  } else {
    cout << "No solution exists." << endl;
    return false;
  }
}
```

Variant 17.8.1: Compute a placement of eight queens on an 8×8 chessboard in which no two queens attack each other.

Variant 17.8.2: Compute a placement of 32 knights, or 14 bishops, 16 kings or eight rooks on an 8×8 chessboard in which no two pieces attack each other.

Variant 17.8.3: Compute the smallest number of queens that can be placed to attack each uncovered square.

Problem 17.9, *pg. 142*: *Given an array of digits A and a nonnegative integer k, intersperse multiplies ($\times$) and adds ($+$) with the digits of A such that the resulting arithmetical expression evaluates to k. For example, if A is $\langle 1,2,3,2,5,3,7,8,5,9 \rangle$ and k is 995, then k can be realized by the expression "$123 + 2 + 5 \times 3 \times 7 + 85 \times 9$".*

Solution 17.9: Let A be the array of n digits and k the target sum. We want to intersperse $\times$ and $+$ operations among these characters in such a way that the resulting expression equals k.

For each pair of characters, $(A[i], A[i+1])$, we can choose to insert a $\times$, a $+$, or no operator. The number of such locations is $n-1$, implying we can encode the choice with an array of length $n-1$. Each entry is one of three values—$\times$, $+$, and $\sqcup$ (which indicates no operator is added at that location). Exactly 3^{n-1} such arrays exist, so a brute-force solution is to systematically enumerate all arrays. For each enumerated array, we compute the resulting expression, and return as soon as we evaluate to k. The time complexity is $O(n3^n)$, since each expression takes time $O(n)$ to evaluate.

The performance can be improved heuristically using pruning. For example, if we have a partial assignment that inserts $+$ between i and $i+1$, and an assignment of operators to $A[0:i]$ yields an expression that evaluates to k', then we need to search for an assignment of operators to $A[i+1:n-1]$ that yields $k-k'$. (Note we cannot do similar pruning with $\times$ because $\times$ has a higher precedence.) Additional pruning can be based on the observation that the maximum value of an expression corresponds to the case where no operators are inserted. For the previous example, if $k-k'$ is greater than the integer encoded by digits in $A[i+1:n-1]$, we can stop searching. Here is an implementation in C++; it makes heavy use of the STL.

```
int evaluate(list<int> operand_list, const list<char> &oper_list) {
  // Evaluate '*' first
  auto operand_it = operand_list.begin();
  for (const char &oper : oper_list) {
    if (oper == '*') {
      int product = *operand_it;
      operand_it = operand_list.erase(operand_it);
      product *= *operand_it;
      *operand_it = product;
    } else {
      ++operand_it;
    }
  }

```

```
  // Evaluate '+' second
  return accumulate(operand_list.cbegin(), operand_list.cend(), 0);
}

bool exp_synthesis_helper(const vector<int> &A, const int &k,
                          list<int> &operand_list, list<char> &oper_list,
                          int cur, const int &level) {
  cur = cur * 10 + A[level] - '0';
  if (level == A.size() - 1) {
    operand_list.emplace_back(cur);
    if (evaluate(operand_list, oper_list) == k) {
      auto operand_it = operand_list.cbegin();
      cout << *operand_it++;
      for (const char &oper : oper_list) {
        cout << ' ' << oper << ' ' << *operand_it++;
      }
      cout << " = " << k << endl;
      return true;
    }
    operand_list.pop_back();
  } else {
    // No operator
    if (exp_synthesis_helper(A, k, operand_list, oper_list, cur, level + 1)) {
      return true;
    }

    // Add operator '+'
    operand_list.emplace_back(cur);
    if (k - evaluate(operand_list, oper_list) <=
        stoi(string(A.cbegin() + level + 1, A.cend()))) {  // pruning
      oper_list.emplace_back('+');
      if (exp_synthesis_helper(A, k, operand_list, oper_list, 0, level + 1)) {
        return true;
      }
      oper_list.pop_back();  // revert
    }
    operand_list.pop_back();  // revert

    // Add operator '*'
    operand_list.emplace_back(cur), oper_list.emplace_back('*');
    if (exp_synthesis_helper(A, k, operand_list, oper_list, 0, level + 1)) {
      return true;
    }
    operand_list.pop_back(), oper_list.pop_back();  // revert
  }
  return false;
}

void exp_synthesis(const vector<int> &A, const int &k) {
  list<char> oper_list;
  list<int> operand_list;
  if (exp_synthesis_helper(A, k, operand_list, oper_list, 0, 0) == false) {
    cout << "no answer" << endl;
  }
}
```

Problem 17.10, *pg. 142*: *Given a positive integer n, how would you determine the minimum number of multiplications to evaluate x^n?*

Solution 17.10: It is natural to try divide and conquer, e.g., determine the minimum number of multiplications for each of x^k and $x^{\frac{n}{k}}$, for different values of k. This does not work because the subproblems are not independent—we cannot just add the minimum number of multiplications to compute x^5 and x^6 since both may use x^3.

Instead we resort to branch and bound: we maintain a set of partial solutions which we try to extend to the final solution. The key to efficiency is pruning out partial solutions efficiently.

In our context, a partial solution is a list of exponents that we have already computed. Note that in a minimum solution, we will never have an element repeated in the list. In addition, it suffices to consider partial solutions in which the exponents occur in increasing order since if $k > j$ and x^k occurs before x^j in the chain, then x^k could not be used in the derivation of x^j. Hence we lose nothing by advancing the position of x^k.

Here is code that solves the problem:

```
list<int> get_minimum_expression(const int &n) {
  list<int> init_list;
  init_list.emplace_back(1);

  list<list<int>> exp_lists;
  exp_lists.emplace_back(init_list);
  list<int> min_exp;
  int shortest_size = numeric_limits<int>::max();

  while (exp_lists.empty() == false) {
    list<int> exp = exp_lists.front();
    exp_lists.pop_front();
    // Try all possible combinations in a list
    for (const int &i : exp) {
      for (const int &j : exp) {
        int sum = i + j;
        if (shortest_size > exp.size() + 1) {
          if (sum == n) {
            min_exp = exp;
            min_exp.emplace_back(sum);
            shortest_size = exp.size() + 1;
          } else if (sum < n && sum > exp.back()) {
            list<int> ext = exp;
            ext.emplace_back(sum);
            exp_lists.emplace_back(ext);
          }
        }
      }
    }
  }
  return min_exp;
```

```
}
```

If $n = 30$, the code runs in a fraction of a second. It reports $\langle x, x^2, x^3, x^5, x^{10}, x^{15}, x^{30} \rangle$. In all, 7387 partial solutions are examined.

Other bounding techniques are possible. For example, from the binary representation of 30 (11110), we know that seven multiplications suffice (compute x^2, x^4, x^8, x^{16}, and then multiply these together).

The code could avoid considering all pairs i, j and focus on pairs that just involve the last element since other pairs will have been considered previously. More sophisticated bounding can be applied: a chain like $\langle x, x^2, x^3, x^6, x^7 \rangle$ will require at least three more multiplications. The reason is k multiplications starting at a^x yield a maximum of $a^{x2^k} = a^{x \cdot 2^k}$, and $a^{x \cdot 2^k} \geq a^y$ iff $x \cdot 2^k \geq y$. Dividing by x and taking logs, we see $k \geq \lg \frac{y}{x}$. In particular, $\lg \frac{30}{7} > 2.099$ and so this chain can be safely pruned. When selecting a partial solution to continue searching from, we could choose one that is promising, e.g., the shortest solution—this might lead to better solutions faster and therefore more bounding on other search paths.

For hand calculations, these techniques are important but are trickier to code and our original code solves the given problem reasonably quickly.

Problem 17.11, *pg. 143*: *Design an algorithm for checking if a CNF expression is satisfiable.*

Solution 17.11: A reasonable way to proceed is to use backtracking. We choose a variable v, see if there exists a satisfying assignment when $v = 0$. If no such assignment exists, we try $v = 1$. If no satisfying assignment exists for $v = 0$ and for $v = 1$, the expression is not satisfiable.

Once we choose a variable and set its value, the expression simplifies—we need to remove clauses where v appears if we set $v = 1$ and remove clauses where v' appears when we set $v = 0$. In addition, whenever we get to a unit clause—one where a single literal appears—we know that in any satisfying assignment for the current expression, that literal must be set to true; this rule leads to additional simplification. Conversely, if all the clauses are true, we do not need to proceed further—every assignment to the remaining variables makes the expression true.

Variables may be chosen in various ways. One natural choice is to pick the variable which appears the most times in clauses with two literals since it leads to the most unit clauses on simplification. Another choice is to pick the variable which is the most binate—i.e., it appears the most times in negated and non-negated forms.

Problem 17.12, *pg. 143*: *How would you test the Collatz conjecture for the first n positive integers?*

Solution 17.12: Often interview questions are open-ended with no definite good solution—all you can do is provide a good heuristic and code it well. For the Collatz conjecture, the general idea is to iterate through all numbers and for each number repeatedly apply the rules till you reach 1. Here are some of the ideas that you can try to accelerate the check:

- Reuse computation by storing all the numbers you have already proven to converge to 1; that way, as soon as you reach such a number, you can assume it would reach 1.
- To save space, restrict the hash table to odd numbers.
- If you have tested every number up to k, you can stop the chain as soon as you reach a number that is less than or equal to k. You do not need to store the numbers below k in the hash table.
- If multiplication and division are expensive, use bit shifting and addition.
- Partition the search set and use many computers in parallel to explore the subsets, as show in Solution 18.11 on Page 410.

Since the numbers in a sequence may grow beyond 32 bits, you should use 64-bit integer and keep testing for overflow; alternately, you can use arbitrary precision integers.

```
bool test_Collatz_conjecture(const int &n) {
  // Stores the odd number that converges to 1
  unordered_set<long> table;

  // Start from 2 since we don't need to test 1
  for (int i = 2; i <= n; ++i) {
    unordered_set<long> sequence;
    long test_i = i;
    while (test_i != 1 && test_i >= i) {
      // A cycle means Collatz fails.
      if (sequence.emplace(test_i).second == false) {
        return false;
      }

      if (test_i & 1) {  // odd number
        if (table.emplace(test_i).second == false) {
          break;  // this number have already be proven to converge to 1
        }
        long next_test_i = 3 * test_i + 1;  // 3n + 1
        if (next_test_i <= test_i) {
          throw overflow_error("test process overflow");
        }
        test_i = next_test_i;
      } else {  // even number
        test_i >>= 1;  // n / 2
      }
    }
    table.erase(i);  // removes i from table
  }
  return true;
}
```

Problem 17.13, *pg. 143*: *You need to schedule n lectures in m classrooms. Some of those lectures are prerequisites for others. All lectures are one hour-long and start on the hour. How would you choose when and where to hold lectures to finish all the lectures as soon as possible?*

Solution 17.13: We are given a set of n unit duration lectures and m classrooms. The lectures can be held simultaneously as long as no two lectures need to happen in the same classroom at the same time and all the precedence constraints are met.

The problem of scheduling these lectures to minimize the time taken to completion is known to be NP-complete. (The same problem with an unlimited number of classrooms can be solved in polynomial time; it is the subject of Problem 16.8 on Page 136.)

This problem is naturally modeled using graphs. We model lectures as vertices, with an edge from vertex u to vertex v if u is a prerequisite for v. Clearly, the graph must be acyclic for the precedence constraints to be satisfied.

If only one lecture room exists, we can simply hold the lectures in topological order and complete the n lectures in n time (assuming each lecture is of unit duration).

We can develop heuristics using the following observation. Suppose at a given time S is a set of lectures whose precedence constraints have been satisfied. If the cardinality of S is less than or equal to m, we can schedule all the lectures in S; otherwise, we need to schedule a subset.

Subset selection can be based on several heuristic criterion.

- Rank order lectures based on the length of the longest dependency chain that they are at the start of.
- Rank order lectures based on the number of lectures that they are immediate prerequisites for.
- Rank order lectures based on the total number of lectures that they are direct or indirect prerequisites for.

We can also use combinations of these criteria to order the lectures that are currently schedulable.

If the candidate set is less than size m, we schedule all the lectures; otherwise, we choose the m most critical lectures and schedule those—the idea is that they should be scheduled sooner since they are at the start of longer dependency chains.

Problem 18.1, *pg. 145*: *Design an online spell correction system. It should take as input a string s and return an array of entries in its dictionary which are closest to the string using the Levenshtein distance specified in Problem 15.11 on Page 120. Cache the most recently computed result.*

Solution 18.1: The naïve solution would be:

```
public class S1 extends SpellCheckService {
  static String wLast = null;
  static String [] closestToLastWord = null;

  public static void service(ServiceRequest req, ServiceResponse resp) {
    String w = req.extractWordToCheckFromRequest();
    if (!w.equals(wLast)) {
      wLast = w;
      closestToLastWord = Spell.closestInDictionary(w);
    }
    resp.encodeIntoResponse(closestToLastWord);
```

```
  }
}
```

This solution has a race condition. Suppose Threads A and B run the service. Suppose Thread A updates `wLast`, and then Thread B is scheduled. Now Thread B reads `wLast` and `closestToLastWord`. Since Thread A has not updated `closestToLastWord`, if `wLast` equals the check string `w` passed to B, the cached `closestToLastWord` B returns corresponds to the previous value of `wLast`. The call to `closestToLastWord` could take quite long or be very fast, depending on the length and contents of `checkWord`. Hence it is entirely possible that Thread B reads both `wLast` and `closestToLastWord` between Thread A's updates them.

A thread-safe solution would be to declare `service` to be synchronized; in this case, only one thread could be executing the method and there is no race between write to `wLast` and `closestToLastWord`. This leads to poor performance—only one thread can be executing at a time.

The solution is to lock just the part of the code that operates on the cached values—specifically, the check on the cached value and the updates to the cached values:

```
public class S2 extends SpellCheckService {
  static String wLast = null;
  static String [] closestToLastWord = null;

  public static void service(ServiceRequest req, ServiceResponse resp) {
    String w = req.extractWordToCheckFromRequest();
    String [] result  = null;
    synchronized (S2.class) {
      if (w.equals(wLast)) {
        result = Arrays.copyOf(closestToLastWord, closestToLastWord.length);
      }
    }
    if (result == null) {
      result = Spell.closestInDictionary(w);
      synchronized (S2.class) {
        wLast = w;
        closestToLastWord = result;
      }
    }
    resp.encodeIntoResponse(result);
  }
}
```

In the above code, multiple threads can be in their call to `closestInDictionary` which is good because the call may take a long time. Locking ensures that the read assignment on a hit and write assignment on completion are atomic. Note that we have to clone `closestToLastWord` when assigning to `result` since otherwise, `closestToLastWord` might change before we encode it into the response.

Variant 18.1.1: Threads 1 to n execute a method called `critical`. Before this, they execute a method called `rendezvous`. The synchronization constraint is that only one thread can execute `critical` at a time, and all threads must have completed

executing `rendezvous` before `critical` can be called. You can assume n is stored in a variable n that is accessible from all threads. Design a synchronization mechanism for the threads. All threads must execute the same code. Threads may call `critical` multiple times, and you should ensure that a thread cannot call `critical` a $(k+1)$-th time until all other threads have completed their k-th calls to `critical`.

Variant 18.1.2: In this problem you are to design a synchronization mechanism for a pool. This is a data structure that combines requests. Specifically, requests come from two types of threads. The pool has a capacity of four requests. A thread cannot have more than one request in the pool. When the pool is full, it must be the case that requests from both types of threads are present. Exactly one of the requesting threads must call the `launch` function when four requests are in the pool. Each thread corresponding to a request in the pool should invoke a `flush` function before `launch` is executed. Threads should call `flush` as late as possible.

Problem 18.2, *pg. 145*: *Suppose you find that the `SimpleWebServer` has poor performance because `processReq` frequently blocks on I/O. What steps could you take to improve `SimpleWebServer`'s performance?*

Solution 18.2: The first attempt to solve this problem might be to have `main` launch a new thread per request rather than process the request itself:

```
class ThreadPerTaskWebServer {
  private static final int SERVERPORT = 8080;
  public static void main(String [] args) throws IOException {
   final ServerSocket serversocket = new ServerSocket(SERVERPORT);
    while (true) {
      final Socket connection = serversocket.accept();
      Runnable task = new Runnable() {
        public void run() {
          Worker.handleRequest(connection);
        }
      };
      new Thread(task).start();
    }
  }
}
```

The problem with this approach is that we do not control the number of threads launched. A thread consumes a nontrivial amount of resources, such as the time taken to start and end the thread and the memory used by the thread. For a lightly-loaded server, this may not be an issue but under load, it can result in exceptions that are challenging, if not impossible, to handle.

The right trade-off is to use a *thread pool*. As the name implies, this is a collection of threads, the size of which is bounded. Java provides thread pools through the `Executor` framework.

```
class TaskExecutionWebServer {
  private static final int NTHREADS = 100;
```

```
  private static final int SERVERPORT = 8080;
  private static final Executor exec = Executors.newFixedThreadPool(NTHREADS);

  public static void main(String[] args) throws IOException {
    ServerSocket serversocket = new ServerSocket(SERVERPORT);
    while (true) {
      final Socket connection = serversocket.accept();
      Runnable task = new Runnable() {
        public void run() {
          Worker.handleRequest(connection);
        }
      };
      exec.execute(task);
    }
  }
}
```

Problem 18.3, *pg. 146*: *Implement a* `Requester` *class. The* `Execute` *method may take an indeterminate amount of time to return; it may never return. You need to have a time-out mechanism for this. Assume* `Requester` *objects have an* `Error` *method that you can invoke.*

Solution 18.3: Our strategy is to launch a thread T per `Requester` object. Thread T in turn launches another thread, S, which calls `execute` and `ProcessResponse`. The call to `execute` in S is wrapped in a try-catch `InterruptedException` loop; if `execute` completes successfully, `ProcessResponse` is called on the result.

After launching S, T sleeps for the timeout interval—when it wakes up, it interrupts S. If S has completed, nothing happens; otherwise, the try-catch `InterruptedException` calls `error`.

Code for this is given below:

```
public String execute(String req, long delay) {
  try {
    // simulate the time taken to perform a computation
    Thread.sleep(delay);
  } catch (InterruptedException e) {
    return error(req);
  }
  return execute(req);
}
public static void Dispatch(final Requestor r, final String request,
                            final long delay) {
  Runnable task = new Runnable() {
    public void run() {
      Runnable actualTask = new Runnable() {
        public void run() {
          String response = r.execute(request, delay);
          r.ProcessResponse(response);
        }
      };
      Thread innerThread = new Thread(actualTask);
      innerThread.start();
```

```
      try {
        Thread.sleep(TIMEOUT);
        innerThread.interrupt();
      } catch(InterruptedException e) {
        e.printStackTrace();
      }
    }
  };
  new Thread(task).start();
}
```

Problem 18.4, *pg. 146*: *Develop a* `Timer` *class that manages the execution of deferred tasks. The* `Timer` *constructor takes as its argument an object which includes a* `Run` *method and a* `name` *field, which is a string.* `Timer` *must support—(1.) starting a thread, identified by name, at a given time in the future; and (2.) canceling a thread, identified by name (the cancel request is to be ignored if the thread has already started).*

Solution 18.4: The two aspects to the design are the data structures and the locking mechanism.

We use two data structures. The first is a min-heap in which we insert key-value pairs: the keys are run times and the values are the thread to run at that time. A dispatch thread runs these threads; it sleeps from call to call and may be woken up if a thread is added to or deleted from the pool. If woken up, it advances or retards its remaining sleep time based on the top of the min-heap. On waking up, it looks for the thread at the top of the min-heap—if its launch time is the current time, the dispatch thread deletes it from the min-heap and executes it. It then sleeps till the launch time for the next thread in the min-heap. (Because of deletions, it may happen that the dispatch thread wakes up and finds nothing to do.)

The second data structure is a hash table with thread ids as keys and entries in the min-heap as values. If we need to cancel a thread, we go to the min-heap and delete it. Each time a thread is added, we add it to the min-heap; if the insertion is to the top of the min-heap, we interrupt the dispatch thread so that it can adjust its wake up time.

Since the min-heap is shared by the update methods and the dispatch thread, we need to lock it. The simplest solution is to have a single lock that is used for all read and writes into the min-heap and the hash table.

Problem 18.5, *pg. 146*: *Implement a synchronization mechanism for the first readers-writers problem.*

Solution 18.5: We want to indicate whether the string is being read as well as whether the string is being written to. We achieve this with a pair of locks—`LR` and `LW` and a read counter locked by `LR`.

A reader proceeds as follows. It locks `LR`, increments the counter, and releases `LR`. After it performs its reads, it locks `LR`, decrements the counter, and releases `LR`. A writer locks `LW`, then performs the following in an infinite loop. It locks `LR`, checks

to see if the read counter is 0; if so, it performs its write, releases LR, and breaks out of the loop. Finally, it releases LW. In the code below we use the Java `wait()` and `notify()` primitives to avoid the CPU cycles wasted in a busy wait.

```
// LR and LW are static members of type Object in the RW class.
// They serve as read and write locks. The static integer
// variable readCount in RW tracks the number of readers.
class Reader extends Thread {
  public void run() {
    while (true) {
      synchronized (RW.LR) {
        RW.readCount++;
        RW.LR.notify();
      }
      System.out.println(RW.data);
      synchronized (RW.LR) {
        RW.readCount--;
        RW.LR.notify();
      }
      Task.doSomeThingElse();
    }
  }
}

class Writer extends Thread {
  public void run() {
    while (true) {
      synchronized (RW.LW) {
      boolean done = false;
      while (!done) {
          synchronized (RW.LR) {
            if (RW.readCount == 0) {
              RW.data = new Date().toString();
              done = true;
            } else {
              // use wait/notify to avoid busy waiting
              try {
                RW.LR.wait();
              } catch (InterruptedException e) {
                System.out.println("InterruptedException in Writer wait");
              }
            }
            RW.LR.notify();
          }
        }
      }
      Task.doSomeThingElse();
    }
  }
}
```

Problem 18.6, *pg. 147*: *Implement a synchronization mechanism for the second readers-writers problem.*

Solution 18.6: We want to give writers the preference. We achieve this by modifying Solution 18.5 on Page 407 to have a reader start by locking `LW` and then immediately releasing `LW`. In this way, a writer who acquires the `LW` lock is guaranteed to be ahead of the subsequent readers.

Problem 18.7, *pg. 147*: *Implement a synchronization mechanism for the third readers-writers problem.*

Solution 18.7: We can achieve fairness between readers and writers by having a bit which indicates whether a read or a write was the last operation done. If the last operation done was a read, a reader on acquiring a lock must release the lock and retry—this gives writers priority in acquiring the lock; a similar operation is done by writers.

Note that this solution entails readers and writers having to wait longer than is absolutely necessary. Specifically, readers may wait even if s is opened for read and writers may wait even if no one else has a lock on s.

Variant 18.7.1: Categorical starvation refers to a phenomenon in which one category of threads make another category of threads wait indefinitely. Both Solutions 18.5 on Page 407 and 18.6 on the preceding page exhibit categorical starvation, with the readers and writers constituting the categories. Solution 18.7 on this page guarantees no categorical starvation. Thread starvation refers to a phenomenon in which a specific thread waits indefinitely while others proceed. Solve Problem 18.7 on Page 147 with the added constraint that it is free of thread starvation.

Problem 18.8, *pg. 147*: *Design a synchronization mechanism for A which ensures that P does not try to add a string into the array if it is full and C does not try to remove data from an empty buffer.*

Solution 18.8: This problem can be solved for a single producer and a single consumer with a pair of semaphores—*fillCount* is incremented and *emptyCount* is decremented whenever an item is added to the buffer. If the producer wants to decrement *emptyCount* when its count is zero, the producer sleeps. The next time an item is consumed, *emptyCount* is incremented and the producer is woken up. The consumer operates analogously. The Java methods, `wait` and `notify`, can be used to implement the desired functionality.

In the presence of multiple producers and consumers, the solution above has two races—two producers can try writing to the same slot and two consumers can read from the same slot. These races can be removed by adding mutexes around the insert and delete calls.

Problem 18.9, *pg. 147*: *Model the barber shop using semaphores and mutexes to ensure correct behavior. Each customer is a thread, as is the barber.*

Solution 18.9: A casual implementation is susceptible to races. For example, a new customer may see the barber cutting hair and go to the waiting room. Before this

customer gets to there, the barber may complete the haircut, check the waiting room, observe it to be empty, and go back to his chair to sleep. This is a form of livelock—the barber and the customer are both idle, waiting for each other. As another example, in the absence of appropriate locking, two customers may arrive simultaneously, see the barber cutting hair, and a single vacant seat in the waiting room, and go to the waiting room to occupy the single chair.

One way to achieve correct operation is to have a single mutex which allows only one person to change state at a time. The barber must acquire the mutex before checking for customers; he must release it when he either begins to sleep or begins to cut hair. A customer must acquire the mutex before entering the shop; he must release it when he sits in either a waiting room chair or the barber chair.

For a complete solution, in addition to the mutex, we need event semaphores to record the number of customers in the waiting room and the number of people getting their hair cut. The event semaphore recording the number of customers in the waiting room is used to wake up the barber when a customer enters; the event semaphore recording the number of customers getting a haircut is used to wake up waiting customers.

Problem 18.10, *pg. 148*: *Implement a synchronization mechanism for the dining philosophers problem.*

Solution 18.10: The natural solution is for each resource to have a lock. The problem arises when each thread i first requests lock i and then lock $i + 1 \bmod n$. Since all locks have already been acquired, the thread deadlocks.

One approach is to have a central controller, which knows exactly which resources are in use and arbitrates conflicting requests. If resources are not available for a thread, the controller can reject its request.

A general principle for avoiding livelock is to order the resources and require that resources be acquired in increasing order and released in decreasing order. For example, if all threads request simultaneously, Resource $n-1$ will be left unrequested (since Thread $n-1$ will request 0 first, and then $n-1$). Thread $n-2$ will then succeed at acquiring Resource $n-1$ since Thread $n-1$ will block on Resource 0.

This solution is not starvation-free, e.g., Thread 2 can wait forever while Threads 1 and 3 alternate. To guarantee that no thread starves, track of the number of times a thread cannot execute when its neighbors release their locks. If this number exceeds some limit, the state of the thread could change to `starving` and the decision procedure to enter the critical section is supplemented to require that none of the neighbors are starving. A philosopher that cannot pick up locks because a neighbor is starving is effectively waiting for the neighbor's neighbor to finish eating. This additional dependency reduces concurrency—raising the threshold for transition to the `starving` state reduces this effect.

Problem 18.11, *pg. 148*: *Design a multi-threaded program for checking the Collatz conjecture. Make full use of the cores available to you. To keep your program from overloading the system, you should not have more than n threads running at a time.*

Solution 18.11: Heuristics for pruning checks on individual integers are discussed in Solution 17.12 on Page 401. The focus on this problem is on implementing a multi-threaded checker. We could have a master thread launch n threads, one per number, starting with $1, 2, \ldots, x$. The master thread would keep track of what number needs to be processed next, and when a thread returned, it could re-assign it the next unchecked number.

The problem with this approach is that the time spent executing the check in an individual thread is very small compared to the overhead of communicating with the thread. The natural solution is to have each thread process a subrange of $[1, U]$. We could do this by dividing $[1, U]$ into n equal sized subranges, and having Thread i handle the i-th subrange.

The heuristics for checking the Collatz conjecture take longer on some integers than others, and in the strategy above there is the potential of a situation arising where one thread takes much longer to complete than the others, which leads to most of the cores being idle.

A good compromise is to have threads handle smaller intervals, which are still large enough to offset the thread overhead. We can maintain a work-queue consisting of unprocessed intervals, and assigning these to returning threads. The Java Executor framework is ideally suited to implementing this, and an implementation is given in the code below:

```
// Performs basic unit of work
class MyRunnable implements Runnable {
  public int lower;
  public int upper;

  MyRunnable(int lower, int upper) {
    this.lower = lower;
    this.upper = upper;
  }

  @Override
  public void run() {
    for (int i = lower; i <= upper; ++i) {
      Collatz.CollatzCheck(i, new HashSet<BigInteger>());
    }
    System.out.println("(" + lower + "," + upper + ")" );
  }
}

public class Collatz {
  // Checks an individual number
  public static boolean CollatzCheck(BigInteger x, Set<BigInteger> visited) {
    if (x.equals(BigInteger.ONE)) {
      return true;
    } else if (visited.contains(x)) {
      return false;
    }
    visited.add(x);
    if (x.getLowestSetBit() == 1) {  // odd number
```

```
      return CollatzCheck(
          (new BigInteger("3")).multiply(x).add(BigInteger.ONE), visited);
    } else {  // even number
      return CollatzCheck(x.shiftRight(1), visited); // divide by 2
    }
  }

  public static boolean CollatzCheck(int x, Set<BigInteger> visited) {
    BigInteger b = new BigInteger(new Integer(x).toString());
    return CollatzCheck(b, visited);
  }

  public static ExecutorService execute() {
    // Uses the Executor framework for task assignment and load balancing
    List<Thread> threads = new ArrayList<Thread>();
    ExecutorService executor = Executors.newFixedThreadPool(NTHREADS);
    for (int i = 0 ; i < (N / RANGESIZE); ++i) {
      Runnable worker = new MyRunnable(i * RANGESIZE + 1,
                                       (i + 1) * RANGESIZE);
      executor.execute(worker);
    }
    executor.shutdown();
    return executor;
  }
}
```

Problem 18.12, *pg. 148*: *Design an algorithm that computes the sequence of transfers that minimizes the time taken to transfer a message from the root to all the nodes in the tree.*

Solution 18.12: We solve this problem using a straightforward bottom-up recursion. Let $T(u)$ denote the minimum number of seconds to propagate the message to the subtree rooted at u. Note that $T(u) = 0$ if u is a leaf.

Suppose u is not a leaf. Order u's children $v_0, v_1, \ldots, v_{k-1}$ by decreasing transfer times. It is straightforward to see it is optimum for u to send the messages in this order. Therefore $T(u) = \max_{0 \le i \le n-1}(T(v_i) + i + 1)$. The run time is dominated by the time to sort results from child nodes, leading to an $O(n \log n)$ time bound, where n is the number of nodes in the tree.

Problem 18.13, *pg. 148*: *Devise a protocol by which hosts can elect a leader from the set of all hosts participating in the protocol. The protocol should be fast, in that it converges quickly; it should be efficient, in that it should use few connections and small messages.*

Solution 18.13: Think of the hosts as being vertices in a directed graph with an edge from A to B, if A initially know B's IP address.

We will study variants of this problem—synchronized or unsynchronized hosts, and known or unknown bounds on the number of hosts. We will compare solutions with respect to convergence time, message size, and the number of messages. We assume the graph is weakly connected (otherwise the problem is unsolvable).

First, assume that the hosts are all synchronized to a common clock (there are

standard protocols which can allow computers to synchronize within a few tens of milliseconds; alternately, Global Positioning System (GPS) signals can be used to achieve even tighter synchronization).

We first consider the case where the number of hosts n and the diameter D of the network is known to all the hosts. Our algorithm will elect the host with the highest IP address as the leader.

Since hosts are synchronized, we can proceed in rounds. The simplest algorithm for leader election is flooding—each host keeps track of the highest IP address it knows about; the highest IP address is initialized to its own IP address. In each round, host propagates the highest IP address it knows of to each of its (initial) neighbors. After D rounds, if the highest IP address a host knows of is its own, it declares itself the leader.

Here is a small improvement to this algorithm to reduce the number of messages sent—a host sends out an update only when the highest IP address it knows about changes.

It takes D rounds to converge and Dm messages are communicated, where m is the number of edges in the graph. The number of iterations to convergence can be reduced to $\lg D$ by having each host send the set of hosts it has discovered in each iteration to each host it knows about. This leads to faster convergence since the distance to the frontier of undiscovered hosts doubles in each iteration. However it requires much more communication—the final round involves n hosts sending n messages and each message has the ids of n hosts. Furthermore, unlike the original algorithm, this variant requires messages to potentially traverse longer routes (in the original algorithm, a host communicated only with the hosts it knew about initially). The algorithm works correctly even if D is just an upper bound on the true diameter.

When n and D are completely unknown, leader election can be performed through a distributed BFS. Each host starts by sending out a search message to all of its outgoing neighbors. In any round, if a host receives a search message, it chooses one of the hosts from which it received a search as its parent and informs its parent about its selection. (Since we are assuming an IP network, a child can directly communicate its selection back to its parent.)

This procedure constructs a BFS tree for each host. Completion can be detected by having hosts respond to search messages with both a parent or non-parent message as well as a notification of completion from its children. When BFS completes, each host has complete knowledge of the graph and can determine the leader.

Now, we consider the asynchronous case. The flooding algorithms we considered earlier cannot be directly generalized to asynchronous hosts because there is no notion of a round. However we can simulate rounds by having hosts tag their messages with the round number. A host waits to receive all round r messages from all its neighbors before performing its round r update. This algorithm cannot avoid sending messages if the highest IP it knows about does not change in round r since the neighbors depend on receiving all their round r messages before they can advance.

e-Variant 18.13.1: Devise a protocol by which a collection of hosts on the Internet can discover each other.

Variant 18.13.2: A set of soldiers is arranged in a line. All soldiers who are not at the two ends have a copy of the same finite state machine; the soldiers at the ends may have different finite state machines. The finite state machines operate in lock-step, i.e., they all update on a common clock. Each of these finite state machines has at least three states—quiescent, excited, and firing. Design the finite state machines so that when all soldiers except for one at an end begin in a quiescent state, and the remaining soldier is in the excited state, all soldiers will enter the firing state at the same instant at some time in the future.

Problem 18.14, *pg. 149*: *How would you sort a billion* 1000 *byte strings? How about a trillion* 1000 *byte strings?*

Solution 18.14: A billion 1000 byte strings cannot fit in the RAM of a single machine, but can fit on the hard drive of a single machine. Therefore, one approach is to partition the data into smaller blocks that fit in RAM, sort each block individually, write the sorted block to disk, and then combine the sorted blocks. The sorted blocks can be merged using for example Solution 10.1 on Page 248. The UNIX `sort` program uses these principles when sorting very large files, and is faster than direct implementations of the merge-based algorithm just described.

If the data consists of a trillion 1000 byte strings, it cannot fit on a single machine—it must be distributed across a cluster of machines. The most natural interpretation of sorting in this scenario is to organize the data so that lookups can be performed via binary search. Sorting the individual datasets is not sufficient, since it does not achieve a global ordering—lookups entail a binary search on each machine. The straightforward solution is to have one machine merge the sorted datasets, but then that machine will become the bottleneck.

A solution which does away with the bottleneck is to first reorder the data so that the i-th machine stores strings in a range, e.g., Machine 3 is responsible for strings that lie between *daily* and *ending*. The range-to-machine mapping R can be computed by sampling the individual files and sorting the sampled values. If the sampled subset is small enough, it can be sorted by a single machine. The techniques in Solution 11.13 on Page 89 can be used to determine the ranges assigned to each machine. Specifically, let A be the sorted array of sampled strings. Let there be n machines. Define $r_i = iA[|A|/n]$. Then Machine i is responsible for strings in the range $[r_i, r_{i+1})$. If the distribution of the data is known *a priori*, e.g., it is uniform, the sampling step can be skipped.

The reordering can be performed in a completely distributed fashion by having each machine route the strings it begins with to the responsible machines.

After reordering, each machine sorts the strings it stores. Consequently queries such as lookups can be performed by using R to determine which individual machine to forward the lookup to.

Problem 18.15, *pg. 149*: *Implement crawling under the constraint that in any given minute your crawlers do not request more than b bytes from any website.*

Solution 18.15: This problem, as posed, is ambiguous.

- Since we usually download one file in one request, if a file is greater than b bytes, there is no way we can meet the constraint of serving fewer than b bytes every minute, unless we can work with the lower layers of networking stack such as the transport layer or the network layer. Often the system designer could look at the distribution of file sizes and conclude that this problem happens so infrequently that we do not care. Alternately, we may choose to download no more than the first b bytes of any file.
- Given that the host's bandwidth is a resource for which there could be contention, one important design choice to be made is how to resolve a contention. Do we let requests get served in first-come first-served order or is there a notion of priority? Often crawlers have a built-in notion of priority based on how important the document is to the users or how fresh the current copy is.

One way of doing this could be to maintain a permission server with which each crawler checks to see if it is okay to hit a particular host. The server can keep an account of how many bytes have been downloaded from the server in the last minute and not permit any crawler to hit the server if we are already close to the quota. If we do not care about priority, then we can keep the interface synchronous where a server requests permission to download a file and it immediately gets approved or denied. If we care about priorities, then the server may enqueue the request and inform the crawler when it is alright to download the file. The queues at the permission server may be based on priorities.

If the permission server becomes a bottleneck, we can use multiple permission servers such that the responsibility of a given host is decided by applying a hash function to the host name and assigning it to a particular server based on the hash code.

Problem 19.1, *pg. 150*: *Design a program that produces high quality mosaics with minimal compute time.*

Solution 19.1: A good way to begin is to partition the image into $s \times s$-sized squares, compute the average color of each such image square, and then find the tile that is closest to it in the color space. Distance in the color space can be the $L2$-distance over the Red-Green-Blue (RGB) intensities for the color. As you look more carefully at the problem, you might conclude that it would be better to match each tile with an image square that has a similar structure. One way could be to perform a coarse pixelization (2×2 or 3×3) of each image square and finding the tile that is "closest" to the image square under a distance function defined over all pixel colors. In essence, the problem reduces to finding the closest point from a set of points in a k-dimensional space.

Given m tiles and an image partitioned into n squares, then a brute-force approach would have $O(mn)$ time complexity. You could improve on this by first indexing the

tiles using an appropriate search tree. You can also run the matching in parallel by partitioning the original image into subimages and searching for matches on the subimages independently.

Problem 19.2, *pg. 150*: *Given a million documents with an average size of* 10 *kilobytes, design a program that can efficiently return the subset of documents containing a given set of words.*

Solution 19.2: The predominant way of doing this is to build inverted indices. In an inverted index, for each word, we store a sequence of locations where the word occurs. The sequence itself could be represented as an array or a linked list. Location is defined to be the document ID and the offset in the document. The sequence is stored in sorted order of locations (first ordered by document ID, then by offset). When we are looking for documents that contain a set of words, what we need to do is find the intersection of sequences for each word. Since the sequences are already sorted, the intersection can be done in time proportional to the aggregate length of the sequences. We list a few optimizations below.

- *Compression*—compressing the inverted index helps both with the ability to index more documents as well as memory locality (fewer cache misses). Since we are storing sorted sequences, one way of compressing is to use delta compression where we only store the difference between the successive entries. The deltas can be represented in fewer bits.
- *Caching*—the distribution of queries is usually fairly skewed and it helps a great deal to cache the results of some of the most frequent queries.
- *Frequency-based optimization*—since search results often do not need to return every document that matches (only top ten or so), only a fraction of highest quality documents can be used to answer most of the queries. This means that we can make two inverted indices, one with the high quality documents that stays in RAM and one with the remaining documents that stays on disk. This way if we can keep the number of queries that require the secondary index to a small enough number, then we can still maintain a reasonable throughput and latency.
- *Intersection order*—since the total intersection time depends on the total size of sequences, it would make sense to intersect the words with smaller sets first. For example, if we are looking for "USA GDP 2009", it would make sense to intersect the lists for GDP and 2009 before trying to intersect the sequence for USA.

We could also build a multilevel index to improve accuracy on documents. For a high priority web page, we can decompose the page into paragraphs and sentences, which are indexed individually. That way the intersections for the words might be within the same context. We can pick results with closer index values from these sequences. See the sorted array intersection problem 13.5 on Page 99 and digest problem 12.14 on Page 96 for related issues.

Problem 19.3, *pg. 151*: *You are given a large set of strings S. Given a query string Q, how*

would you design a system that can quickly identify the longest string $p \in S$ that is a prefix of Q?

Solution 19.3: This is a well studied problem because of its implications to building a high speed Internet backbone. A number of approaches have been proposed and used in IP routers. One simple approach is to build a trie data structure such that we can traverse the trie for an IP address till we hit a node that has a label. This essentially requires one pointer indirection per bit of input. The lookup speed can be improved a little at the cost of memory by making fatter nodes in the trie that consume multiple bits at a time. See Solution 9.14 on Page 247 for more details.

A number of approaches have been tried in software and hardware to speed the lookup process:

- Binary search on hash tables—we can have one hash table for each possible prefix length and then do a search for the longest matching prefix by searching through the hash tables. This can take as many as 32 hash table lookups in the worst case. One way to reduce the number of lookups is to perform binary search for the longest matching prefix. For binary search to work, we have to insert additional prefixes in the hash tables to ensure that if a longer prefix exists, binary search does not terminate early. This can be done by performing a binary search for each prefix and inserting additional dummy entries wherever the binary search terminates early. This could inflate the size of hash tables by a factor of $\lg 32 = 5$; in practice the blowup is much smaller.
- Ternary Content Addressable Memory (TCAM). In a conventional RAM, the user supplies an address, and the RAM outputs the data stored at that address. In a Content Addressable Memory (CAM), the user supplies a key, and the CAM returns a Boolean indicating if the key is stored at any address. Depending on the CAM, it may also return the lowest address that stores that key, and possibly a corresponding value. A TCAM is a specialized CAM, where instead of storing 0s and 1s, a single unit of memory can also store a third state called the "don't care" state. The contents of memory can be addressed by partial contents of the memory. A TCAM where each memory location stores 32 ternary values can be used to store prefixes. Each prefix is padded with "don't care" bits to make it 32 ternary values. This way, when we use an IP address to address the TCAM, we get all the matching prefixes. If we store longer prefixes at lower memory locations, the TCAM will return the longest matching prefixes.

Problem 19.4, *pg. 151*: *How would you build a spelling correction system?*

Solution 19.4: The basic idea behind most spelling correction systems is that the misspelled word's Levenshtein distance from the intended word tends to be very small (one or two edits). Hence if we keep a hash table for all the words in the dictionary and look for all the words that are within two Levenshtein distances of the text, most likely, the intended word will be found in this set. If the alphabet has m characters and the search text has n characters, we would need to perform roughly nm^2 hash table lookups. The intersection of set of all strings at a distance of two or

less from a word and the set of dictionary words may be large. It is important to provide a ranked list of suggestions to the users, with the most likely candidates are at the beginning of the list. There are several ways to achieve this.

- Typing errors model—often spelling mistakes are a result of typing errors. Typing errors can be modeled based on keyboard layouts.
- Phonetic modeling—a big class of spelling errors happen when the person spelling it knows how the words sounds but does not know the exact spelling. In such cases, it helps to map the text to phonemes and then find all the words that map to the same phonetic sequence.
- History of refinements—often users themselves provide a great amount of data about the most likely misspellings by first entering a misspelled word and then correcting it. This historic data is often immensely valuable for spelling correction.
- Stemming—often the size of a dictionary can be reduced by keeping only the stemmed version of each word. (This entails stemming the query text.)

Problem 19.5, ***pg. 151***: *Design a stemming algorithm that is fast and effective.*

Solution 19.5: Stemming is a large topic. Here we mention some basic ideas related to stemming, however this is in no way a comprehensive discussion on stemming approaches.

Most stemming systems are based on simple rewrite rules, e.g., remove suffixes of the form "es", "s", and "ation". Suffix removal does not always work. For example, wolves should be stemmed to wolf. To cover this case, we may have a rule that replaces the suffix "ves" with "f".

Most rules amount to matching a set of suffixes and applying the corresponding transformation to the string. One way of efficiently performing this is to build a finite state machine based on all the rules.

A more sophisticated system might have exceptions to the broad rules based on the stem matching some patterns. The Porter stemmer, developed by Martin Porter, is considered to be one of the most authoritative stemming algorithms in the English language. It defines several rules based on patterns of vowels and consonants.

Other approaches include the use of stochastic methods to learn rewrite rules and n-gram based approaches where we look at the surrounding words to determine the correct stemming for a word.

Problem 19.6, ***pg. 152***: *How would you implement TEX?*

Solution 19.6: Note that the problem does not ask for the design of TEX, which itself is a complex problem involving feature selection, and language design. There are a number of issues common to implementing any such program: programming language selection, lexing and parsing input, error handling, macros, and scripting.

Two key implementation issues specific to TEX are a specifying fonts and symbols (e.g., A, b, f, $\sum$, $\oint$, $\uplus$), and assembling a document out of components.

Focusing on the second aspect, a reasonable abstraction is to use a rectangular bounding box to describe components. The description is hierarchical: each individual symbol is a rectangle, lines and paragraphs are made out of these rectangles and are themselves rectangles, as are section titles, tables and table entries, and included images. A key algorithmic problem is to assemble these rectangles, while preserving hard constraints on layout, and soft constraints on aesthetics. See also Problem 15.13 on Page 121 for an example of the latter.

Turning our attention to symbol specification, the obvious approach is to use a 2D array of bits to represent each symbol. This is referred to as a bit-mapped representation. The problem with bit-mapped fonts is that the resolution required to achieve acceptable quality is very high, which leads to huge documents and font-libraries. Different sizes of the same symbol need to be individually mapped, as do italicized and bold-face versions.

A better approach is to define symbols using mathematical functions. A reasonable approach is to use a language that supports quadratic and cubic functions, and elementary graphics transformations (rotation, interpolation, and scaling). This approach overcomes the limitations of bit-mapped fonts—parameters such as aspect ratio, font slant, stroke width, serif size, etc. can be programmed.

Other implementation issues include enabling cross-referencing, automatically creating indices, supporting colors, and outputting standard page description formats (e.g., PDF)).

Donald Knuth's book *"Digital Typography"* describes in great detail the design and implementation of TEX.

Problem 19.7, *pg. 152*: *Implement the UNIX* `tail` *command.*

Solution 19.7: The natural approach to this problem is to read the input one line at a time. Each line can be stored in a queue—when the queue size is equal to the number of desired lines, each additional line is inserted at the tail, and the line at the head is deleted. (A circular buffer is a particularly appropriate implementation for this application.) The drawback of this approach is that it entails reading the entire file which could be huge.

The UNIX OS provides the ability to perform random access on a file, essentially allowing us to treat the file as an array of characters, albeit with much slower access times. This capability is exposed to C++ programmers through the `seekg` ("seek get") function in the `istream` library. In the code below, we use `seekg` to process the file in reverse order starting the end of the file. We store the characters in a string, stopping when the specified number of lines have been read.

```
string tail(const string &file_name, const int &tail_count) {
  fstream file_ptr(file_name.c_str());

  file_ptr.seekg(0, ios::end);
  int file_size = file_ptr.tellg(), newline_count = 0;
  string output;  // stores the last tail_count lines
  // Reads file in reverse looking for '\n'
  for (int i = 0; i < file_size; ++i) {
```

```
    file_ptr.seekg(file_size - i - 1, ios::beg);
    char c;
    file_ptr.get(c);
    if (c == '\n') {
      ++newline_count;
      if (newline_count > tail_count) {
        break;
      }
    }
    output.push_back(c);
  }
  // Reverse the output string using the reverse function
  // from the <algorithm> library in STL. The arguments
  // are iterators to the start and end of String object.
  reverse(output.begin(), output.end());
  return output;
}
```

Problem 19.8, *pg. 152*: *Design a feature that allows a studio to enter a set V of videos that belong to it, and to determine which videos in the YouTV.com database match videos in V.*

Solution 19.8: If we replaced videos everywhere with documents, we could use the techniques in Solution 12.13 on Page 286, where we looked for near duplicate documents by computing hash codes for each length-k substring.

Videos differ from documents in that the same content may be encoded in many different formats, with different resolutions, and levels of compression.

One way to reduce the duplicate video problem to the duplicate document problem is to re-encode all videos to a common format, resolution, and compression level. This in itself does not mean that two videos of the same content get reduced to identical files—the initial settings affect the resulting videos. However, we can now "signature" the normalized video.

A trivial signature would be to assign a 0 or a 1 to each frame based on whether it has more or less brightness than average. A more sophisticated signature would be a 3 bit measure of the red, green, and blue intensities for each frame. Even more sophisticated signatures can be developed, e.g., by taking into account the regions on individual frames. The motivation for better signatures is to reduce the number of false matches returned by the system, and thereby reduce the amount of time needed to review the matches.

The solution proposed above is algorithmic. However, there are alternative approaches that could be effective: letting users flag videos that infringe copyright (and possibly rewarding them for their effort), checking for videos that are identical to videos that have previously been identified as infringing, looking at meta-information in the video header, etc.

Variant 19.8.1: Design an online music identification service.

Problem 19.9, *pg. 152*: *Design a system that can compute the ranks of ten billion web pages*

in a reasonable amount of time.

Solution 19.9: Since the web graph can have billions of vertices and it is mostly a sparse graph, it is best to represent the graph as an adjacency list. Building the adjacency list representation of the graph may require a significant amount of computation, depending upon how the information is collected. Usually, the graph is constructed by downloading the pages on the web and extracting the hyperlink information from the pages. Since the URL of a page can vary in length, it is often a good idea to represent the URL by a hash code.

The most expensive part of the PageRank algorithm is the repeated matrix multiplication. Usually, it is not possible to keep the entire graph information in a single machine's RAM. Two approaches to solving this problem are described below.

- Disk-based sorting—we keep the column vector X in memory and load rows one at a time. Processing Row i simply requires adding $A_{i,j}X_j$ to X_j for each j such that $A_{i,j}$ is not zero. The advantage of this approach is that if the column vector fits in RAM, the entire computation can be performed on a single machine. This approach is slow because it uses a single machine and relies on the disk.
- Partitioned graph—we use n servers and partition the vertices (web pages) into n sets. This partition can be computed by partitioning the set of hash codes in such a way that it is easy to determine which vertex maps to which machine. Given this partitioning, each machine loads its vertices and their outgoing edges into RAM. Each machine also loads the portion of the PageRank vector corresponding to the vertices it is responsible for. Then each machine does a local matrix multiplication. Some of the edges on each machine may correspond to vertices that are owned by other machines. Hence the result vector contains nonzero entries for vertices that are not owned by the local machine. At the end of the local multiplication it needs to send updates to other hosts so that these values can be correctly added up. The advantage of this approach is that it can process arbitrarily large graphs.

PageRank runs in minutes on a single machine on the graph consisting of the six million pages that constitute Wikipedia. It takes roughly 70 iterations to converge on this graph. Anecdotally, PageRank takes roughly 200 iterations to converge on the web graph.

Problem 19.10, *pg. 152*: *Design a system for maintaining a set of prioritized jobs that implements the following API:*

1. *Insert a new job with a given priority.*
2. *Delete a job.*
3. *Find the highest priority job.*

Each job has a unique ID. Assume the set cannot fit into a single machine's memory.

Solution 19.10: If we have enough RAM on a single machine, the most simple solution would be to maintain a min-heap where entries are ordered by their priority. An additional hash table can be used to map jobs to their corresponding entry in the min-heap to make deletions fast.

A more scalable solution entails partitioning the problem across multiple machines. One approach is to apply a hash function to the job ids and partition the resulting hash codes into ranges, one per machine. Insert as well as delete require communication with just one server. To do extract-min, we send a lookup minimum message to all the machines, infer the min from their responses, and then delete it.

At a given time many clients may be interested in the the highest priority event, and it is challenging to distribute this problem well. If many clients are trying to do this operation at the same time, we may run into a situation where most clients will find that the min event they are trying to extract has already been deleted. If the throughput of this service can be handled by a single machine, we can make one server solely responsible for responding to all the requests. This server can prefetch the top hundred or so events from each of the machines and keep them in a heap.

In many applications, we do not need strong consistency guarantees. We want to spend most of our resources taking care of the highest priority jobs. In this setting, a client could pick one of the machines at random, and request the highest priority job. This would work well for the distributed crawler application. It is not suited to event-driven simulation because of dependencies.

Problem 19.11, *pg. 153*: *You have guaranteed your clients that 99% of their requests will be serviced in less than one second. How would you design a system to meet this requirement with minimal cost?*

Solution 19.11: Suppose at a given time no more than a fixed number of requests can be served concurrently; pending requests must wait for a slot to open up before they can be served. It is important to queue requests in such a way that the requests that take a long time to serve do not block a large number of short requests behind them.

Suppose the time it takes for the server to process a request is a known easy-to-compute function of the internals of the request, and the service time follows a Pareto distribution. In such cases, the 99-th percentile latency is dramatically reduced by maintaining a short-request queue and a long-request queue. A threshold is used to assign requests to queues. Requests that take longer than the threshold go to the long-request queue; the remainder are assigned to the short-request queue. We pick the threshold to make most requests go to the queue of short requests queue. The requests in the short-request queue are never blocked behind a long-running request. The longer requests do have longer to wait, but overall this strategy is extremely effective at reducing the 99-th percentile latency.

Often the system designer does not know how long a given request will take. Even in this case, it is advantageous to keep two queues. When a request comes in, it is put in the short-request queue. If it takes longer than a certain threshold T_c, it is canceled and added to the long-request queue. Simulation studies and experimentation can be used to derive a suitable choice for T_c.

Problem 19.12, *pg. 153*: *Jingle, a search engine startup, wants to monetize its search results by displaying advertisements alongside search results. Design an online advertising system for Jingle.*

Solution 19.12: Reasonable goals for such a system include

- providing users with the most relevant ads,
- providing advertisers the best possible return on their investment, and
- minimizing the cost and maximizing the revenue to Jingle.

Two key components for such a system are:

- The front-facing component, which advertisers use to create advertisements, organize campaigns, limit when and where ads are shown, set budgets, and create performance reports.
- The ad-serving system, which selects which ads to show on the searches.

The front-facing system can be a fairly conventional web application, i.e., a set of web pages, middleware that responds to user requests, and a database. Key features include:

- User authentication—a way for users to create accounts and authenticate themselves. Alternately, use an existing single sign-on login service, e.g., Facebook or Google.
- User input—a set of form elements to let advertisers specify ads, advertising budget, and search keywords to bid on.
- Performance reports—a way to generate reports on how the advertiser's money is being spent.
- Customer service—even the best of automated systems require occasional human interaction, e.g., ways to override limits on keywords. This requires an interface for advertisers to contact customer service representatives, and an interface for those representatives to interact with the system.

The whole front-end system can be built using, for example, HyperText Markup Language (HTML) and JavaScript. A commonly used approach is to use a LAMP stack on the server-side: Linux as the OS, Apache as the HTTP server, MySQL as the database software, and PHP for the application logic.

The ad-serving system is less conventional. The ad-serving system would build a specialized data structure, such as a decision tree, from the ads database. It chooses ads from the database of ads based on their "relevance" to the search. In addition to keywords, the ad-serving systems can use knowledge of the user's search history, how much the advertiser is willing to pay, the time of day, user locale, and type of browser. Many strategies can be envisioned here for estimating relevance, such as, using information retrieval or machine learning techniques that learn from past user interactions.

The ads could be added to the search results by embedding JavaScript in the results page. This JavaScript pulls in the ads from the ad-serving system directly. This helps isolate the latency of serving search results from the latency of serving ad results.

Problem 19.13, *pg. 153*: *Design a system that automatically generates a sidebar of related articles.*

Solution 19.13: The key technical challenge in this problem is to come up with the list of articles—the code for adding these to a sidebar is trivial.

One suggestion might be to add articles that have proven to be popular recently. Another is to have links to recent news articles. A human reader at Jingle could tag articles which he believes to be significant. He could also add tags such as finance, sports, and politics, to the articles. These tags could also come from the HTML meta-tags or the page title.

We could also provide randomly selected articles to a random subset of readers and see how popular these articles prove to be. The popular articles could then be shown more frequently.

On a more sophisticated level, Jingle could use automatic textual analysis, where a similarity is defined between pairs of articles—this similarity is a real number and measures how many words are common to the two. Several issues come up, such as the fact that frequently occurring words such as "for" and "the" should be ignored and that having rare words such as "arbitrage" and "diesel" in common is more significant than having say, "sale" and "international".

Textual analysis has problems, such as the fact that two words may have the same spelling but completely different meanings (anti-virus means different things in the context of articles on acquired immune deficiency syndrome (AIDS) and computer security). One way to augment textual analysis is to use collaborative filtering—using information gleaned from many users. For example, by examining cookies and timestamps in the web server's log files, we can tell what articles individual users have read. If we see many users have read both A and B in a single session, we might want to recommend B to anyone reading A. For collaborative filtering to work, we need to have many users.

Problem 19.14, *pg. 154*: *Design a driving directions service with a web interface.*

Solution 19.14: At its core, a driving directions service needs to store the map as a graph, where each intersection and street address is a vertex and the roads connecting them are edges. When a user enters a starting address and an ending address, it finds the corresponding vertices and finds the shortest path connecting the two vertices (for some definition of shortest). Issues include:

- Address normalization—a given address may be expressed by the user in different ways, for example, "street" may be shortened to "st", there may not be a city and state, just zip code or vice versa. We need a way to normalize the addresses to a standard format. Sometimes an underspecified address may need to be mapped to some concrete address, for example, a city name to the city center.
- Definition of shortest—different users may have different preferences for routing, for example, shortest distance or fastest path (considering average speed on the road), and avoiding use of freeways. Each of these preferences can be captured by some notion of edge length.
- Approximate shortest distance—given the enormity of a graph representing all the roads in a large country, it would be fairly difficult for a single server to compute the shortest path using standard shortest path algorithms and return in a reasonable amount of time. However using the knowledge that most long

paths go through a standard system of highways and the fact that the vertices and edges in the graph represent points in Euclidean space, we can devise some clever approximation algorithms that run much faster.

The imagery displayed on the web UI can be made out of "tiles"—smaller individual images. JavaScript handlers pull in more tiles when the user requests a zoom or moves to a neighboring region. Tiles can be pre-fetched to improve perceived responsiveness.

Problem 19.15, *pg. 154*: *Design an efficient way of copying one thousand files each* 100 *kilobytes in size from a single lab server to each of* 1000 *servers in a distant data center.*

Solution 19.15: Assume that the bandwidth from the lab machine is a limiting factor. It is reasonable to first do trivial optimizations, such as combining the articles into a single file and compressing this file.

Opening 1000 connections from the lab server to the 1000 machines in the data center and transferring the latest news articles is not feasible since the total data transferred will be approximately 100 gigabytes (without compression).

Since the bandwidth between machines in a data center is very high, we can copy the file from the lab machine to a single machine in the data center and have the machines in the data center complete the copy. Instead of having just one machine serve the file to the remaining 999 machines, we can have each machine that has received the file initiate copies to the machines that have not yet received the file. In theory, this leads to an exponential reduction in the time taken to do the copy.

Several additional issues have to be dealt with. Should a machine initiate further copies before it has received the entire file? (This is tricky because of link or server failures.) How should the knowledge of machines which do not yet have copies of the file be shared? (There can be a central repository or servers can simply check others by random selection.) If the bandwidth between machines in a data center is not a constant, how should the selections be made? (Servers close to each other, e.g., in the same rack, should prefer communicating with each other.)

Finally, it should be mentioned that there are open source solutions to this problem, e.g., Unison and BitTorrent, which would be a good place to start.

Problem 19.16, *pg. 154*: *Design an online poker playing service for Clump Enterprises. Describe both the system architecture and a set of classes.*

Solution 19.16: An online poker playing service would have a front-end system which users interact with and a back-end system which runs the games, manages money, and looks for fraud.

The front-end system would entail a UI for account management—this would cover first-time registration, logging-in, managing online persona, and sending or receiving money. In addition, there would be the game playing UI—this could be as simple as some HTML rendering of the state of the game (cards in hand, cards on the table, bets) and a form to enter a bet. A more sophisticated UI might use JavaScript

to animate the dealing of cards, to change the expression on player's images, and to display status messages.

The back-end needs to form tables of players, shuffle in a truly random manner, deal correctly, check if the player's moves are legal, and update player's finances. It can be implemented using, for example, a Java servlet engine which receives HTTP requests, sends appropriate responses, and updates the database appropriately.

One of the big challenges in such a system is fault-tolerance. On the server side, there exist standard techniques for fault-tolerance, such as replication.

On the client side, there exists the possibility that a player may realize he is in a poor situation and claim that his Internet connection went down. This can be resolved by having a rule that the server will bid on the player's behalf if the player does not respond quickly enough. Another possibility is having the server treat the disconnected player as being in the game but not requiring any more betting of him. This clearly can be abused by the player, so the server needs to record how often a player's connection hangs in a way that is favorable to him.

Collusion among players is another serious problem. Again, the server logs can be mined for examples of players working together to share knowledge of their cards or squeeze other players out. In addition, players can themselves flag suspicious play and customer service representatives can investigate further.

Random number generation is an intensely studied problem but is still often done wrong. A fairly frequent problem is using the ID of the process to seed the random number generator, which means that there are not more than 32768 possible sequences of shuffles on most UNIX systems. This is much less than the 52! possible card sequences.

Now we turn our attention to class design. We begin with the simplest classes: a card class and a deck of cards class. The first has two data members, the suit and the rank, both of which should be enumerated types. The second has a single data field, namely a list of cards; appropriate member functions would be a shuffling routine and a deal n cards function.

Players are modeled by a class that specifies their current hand, and the amount of money they have. Attributes such as name, picture, and location, are best inherited from a user class. Methods include actions to bet/fold, and request cards.

By the rules of poker, a player can bet all that he has left and remain in the game, getting the corresponding share of the pot if he wins. Therefore the pot class should specify how much money has been bet, and who it has been bet by.

The game class itself consists of a deck, a list of players, and a pot. It is responsible for ensuring that players take an appropriate amount of time to play, and that all moves are legal.

Problem 19.17, *pg. 154*: *Design the World Wide Web. Specifically, describe what happens when you enter a URL in a browser address bar, and press return.*

Solution 19.17: At the network level, the browser extracts the domain name component of the URL, and determines the IP address of the server, e.g., through a call to a Domain Name Server (DNS), or a cache lookup. It then communicates using

the HTTP protocol with the server. HTTP itself is built on top of TCP/IP, which is responsible for routing, reassembling, and resending packets, as well as controlling the transmission rate.

The server determines what the client is asking for by looking at the portion of the URL that comes after the domain name, and possibly also the body of the HTTP request. The request may be for something as simple a file, which is returned by the webserver; HTTP spells out a format by which the type of the returned file is specified. For example, the URL http://go.com/imgs/abc.png may encode a request for the file whose hierarchical name is imgs/abc.png relative to a base directory specified at configuration to the web server.

The URL may also encode a request to a service provided by the web server. For example, http://go.com/lookup/flight?num=UA37,city=AUS is a request to the lookup/flight service, with an argument consisting of two attribute-value pair. The service could be implemented in many ways, e.g., Java code within the server, or a Common Gateway Interface (CGI) script written in Perl. The service generates a HTTP response, typically HTML, which is then returned to the browser. This response could encode data which is used by scripts running in the browser. Common data formats include JavaScript Object Notation (JSON) and Extensible Markup Language (XML).

The browser is responsible for taking the returned HTML and displaying it on the client. The rendering is done in two parts. First, a parse tree (the Document Object Model (DOM)) is generated from the HTML, and then a rendering library "paints" the screen. The returned HTML may include scripts written in JavaScript. These are executed by the browser, and they can perform actions like making requests and updating the DOM based on the responses—this is how a live stock ticker is implemented. Styling attributes (Cascading Style Sheets (CSS)) are commonly used to customize the look of a page.

Many more issues exist on both the client and server side: security, cookies, HTML form elements, HTML styling, and handlers for multi-media content, to name a few.

Problem 20.1, *pg. 156*: *Does the following process yield a uniformly random permutation of A? "For $i \in \{0, 1, \ldots, n-1\}$, swap $A[i]$ with a randomly chosen element of A." (The randomly chosen element could be i itself.)*

Solution 20.1: It does not yield all permutations with equal probability. One way to see this is to consider the case $n = 3$. The number of permutations is $3! = 6$. The total number of ways in which we can choose the elements to swap is $3^3 = 27$ and all are equally likely. Since 27 is not divisible by 6, some permutations correspond to more ways than others, ergo not all permutations are equally likely.

The process can be fixed by selecting elements at random and moving them to the end, similar to how we proceeded in Problems 20.2 on Page 156 and 20.7 on Page 157.

Problem 20.2, *pg. 156*: *Let A be an array of n distinct elements. Design an algorithm that returns a subset of k elements of A. All subsets should be equally likely. Use as few calls to the random number generator as possible and use $O(1)$ additional storage. You can return the result in the same array as input.*

Solution 20.2: The problem is trivial when $k = 1$—we simply make one call to the random number generator, take the returned r value mod n. We can swap $A[n-1]$ with $A[r]$; $A[n-1]$ then holds the result.

For $k > 1$, we start by choosing one element at random as above and we now repeat the same process with the $n-1$ element subarray $A[0 : n-2]$. Eventually, the random subset occupies the slots $A[n-k : n-1]$ and the remaining elements are in the first $n-k$ slots.

The algorithm clearly runs in $O(1)$ space. To show that all the subsets are equally likely, we prove something stronger, namely that all permutations of size k are equally likely.

Formally, an m-permutation of a set S of cardinality n is a sequence of m elements of S with no repetitions. It is easily verified that the number of m-permutations is $\frac{n!}{(n-m)!}$.

The induction hypothesis now is that after iteration m, the subarray $A[n-m : n-1]$ contains each possible m-permutation with probability $\frac{(n-m)!}{n!}$.

The base case holds since for $m = 1$, any element is equally likely to be selected.

Suppose the inductive hypothesis holds for $m = l$. Now we study $m = l+1$. Consider a particular $(l+1)$-permutation, say $\langle \alpha_1, \ldots, \alpha_{l+1} \rangle$. This consists of a single element α_1 followed by the l-permutation $\langle \alpha_2, \ldots, \alpha_{l+1} \rangle$. Let E_1 be the event that α_1 is selected in iteration $l+1$ and E_2 be the event that the first l iterations produced $\langle \alpha_2, \ldots, \alpha_{l+1} \rangle$. The probability of $\langle \alpha_1, \ldots, \alpha_{l+1} \rangle$ resulting after iteration $l+1$ is simply $\Pr(E_1 \cap E_2) = \Pr(E_1 \mid E_2)\Pr(E_2)$. By the inductive hypothesis, the probability of permutation $\langle \alpha_2, \ldots, \alpha_{l+1} \rangle$ is $\frac{(n-l)!}{n!}$. The probability $\Pr(E_1|E_2) = \frac{1}{n-l}$ since the algorithm selects from elements in the subarray $A[0 : n-l-1]$ with equal probability. Therefore

$$\Pr(E_1 \cap E_2) = \Pr(E_1|E_2)\Pr(E_2) = \frac{1}{n-l}\frac{(n-l)!}{n!} = \frac{(n-l-1)!}{n!}$$

and induction goes through.

The algorithm generates all random k-permutations with equal probability, from which it follows that all subsets of size k are equally likely.

The algorithm just described makes k calls to the random number generator. When k is bigger than $\frac{n}{2}$, we can optimize by computing a subset of $n-k$ elements to remove from the set. For example, when $k = n-1$, this replaces $n-1$ calls to the random number generator with a single call. Of course, while all subsets are equally likely with this optimization, all permutations are not. Following is the code in C++:

```
template <typename T>
vector<T> offline_sampling(vector<T> A, const int &k) {
  for (int i = 0; i < k; ++i) {
    default_random_engine gen((random_device())());  // random num generator
    // Generate random int in [i, A.size() - 1]
    uniform_int_distribution<int> dis(i, A.size() - 1);
    swap(A[i], A[dis(gen)]);
  }
  A.resize(k);
  return A;
```

```
}
```

Variant 20.2.1: The `rand()` function in the standard C library returns a uniformly random number in $[0, \texttt{RAND_MAX} - 1]$. Does `rand()` mod n generate a number uniformly distributed $[0, n-1]$?

Problem 20.3, *pg. 156*: *Design an algorithm that creates uniformly random permutations of* $\{0, 1, \ldots, n-1\}$. *You are given a random number generator that returns integers in the set* $\{0, 1, \ldots, n-1\}$ *with equal probability; use as few calls to it as possible.*

Solution 20.3: Solution 20.2 on Page 427 can be used with $k = n$. Although the subset that is returned is unique (it will be $\{0, 1, \ldots, n-1\}$), all $n!$ possible orderings of the elements in the set occur with equal probability. (Note that we cannot use the trick to reduce the number of calls to the random number generator at the end of Solution 20.2 on Page 427.)

Problem 20.4, *pg. 156*: *How would you implement a random number generator that generates a random integer i in* $[a, b]$, *given a random number generator that produces either zero or one with equal probability? All generated values should have equal probability. What is the run time of your algorithm, assuming each call to the given random number generator takes* $O(1)$ *time?*

Solution 20.4: Basically, we want to produce a random integer in $[0, b-a]$. Let $l = b - a + 1$. We can produce a random integer in $[0, l-1]$, as follows. Let i be the least integer such that $l \leq 2^i$.

If l is a power of 2, say $l = 2^i$, then all we need are i calls to the 0-1 valued random number generator—the i bits from the calls encode an i bit integer in $[0, l-1]$, and all such numbers are equally likely; so, we can use this integer.

If l is not a power of 2, the i calls may or may not encode an integer in the range 0 to $l-1$. If the number is in the range, we return it; since all the numbers are equally likely, the result is correct.

If the number is outside the range $[0, l-1]$, we try again. The probability of having to try again is less than $\frac{1}{2}$ since $l > 2^{i-1}$. Therefore the probability that we take exactly k steps before succeeding is at most $\frac{1}{2}(1-\frac{1}{2})^{k-1} = \frac{1}{2}^k$. This implies the expected number of trials is less than $1\frac{1}{2} + 2(\frac{1}{2})^2 + 3(\frac{1}{2})^3 + \cdots$. Differentiating the identity $\frac{1}{1-x} = 1 + x + x^2 + x^3 + \cdots$, yields the identity $\frac{1}{(1-x)^2} = 1 + 2x + 3x^2 + 4x^3 + \cdots$. Multiplying both sides by x demonstrate that $\frac{x}{(1-x)^2} = x + 2x^2 + 3x^3 + 4x^4 + \cdots$. Substituting $\frac{1}{2}$ for x in this last identity proves that $1(\frac{1}{2}) + 2(\frac{1}{2})^2 + 3(\frac{1}{2})^3 + \cdots = \frac{\frac{1}{2}}{(1-\frac{1}{2})^2} = 2$. Therefore the expected number of trials is less than 2.

```
int uniform_random_a_b(const int &a, const int &b) {
  int l = b - a + 1, res;
  do {
    res = 0;
    for (int i = 0; (1 << i) < l; ++i) {
```

```
      // zero_one_random is the system-provided random number generator
      res = (res << 1) | zero_one_random();
    }
  } while (res >= 1);
  return res + a;
}
```

Problem 20.5, *pg. 157*: *You are given a set T of n nonnegative real numbers $\{t_0, t_1, \ldots, t_{n-1}\}$ and probabilities $p_0, p_1, \ldots, p_{n-1}$, where $\sum_{i=0}^{n-1} p_i = 1$. Assume that $t_0 < t_1 < \cdots < t_{n-1}$. Given a random number generator that produces values in $[0, 1]$ uniformly, how would you generate a value X from T according to the specified probabilities?*

Solution 20.5: Let $F_X(\alpha)$ be the probability that $X \leq \alpha$, i.e., $F_X(\alpha) = \sum_{t_i \leq \alpha} p_i$. We do the following operation to generate a random value according to X's distribution. Select a number r uniformly at random in the unit interval, $[0, 1]$, then project back from F_X to obtain a value s for X. More specifically, we return the largest t_i such that $F_X(t_i) \leq r$.

By construction, the probability that the value s we return is less than or equal to α is $F_X(\alpha)$, so the distribution for s is exactly the same as that for X.

ϵ-Variant 20.5.1: Given a random number generator that produces values in $[0, 1]$ uniformly, how would you generate a value X from T according to a continuous probability distribution, such as the exponential distribution?

Problem 20.6, *pg. 157*: *Design an algorithm that reads a sequence of packets and maintains a uniform random subset of size k of the read packets when the $n \geq k$-th packet is read.*

Solution 20.6: We store the first k packets. Consequently, we select the n-th packet to add to our subset with probability $\frac{k}{n}$. If we do choose it, we select an element uniformly at random to eject from the subset.

To prove correctness, we use induction on the number of packets that have been read. Specifically, the inductive hypothesis is that all k-sized subsets are equally likely after $n \geq k$ packets have been read.

The number of k-size subsets is $\binom{n}{k}$, implying the probability of any k-size subset should be $\frac{1}{\binom{n}{k}}$.

For the base case, $n = k$, there is exactly one subset of size k which is what the algorithm computes.

Assume the induction hypothesis holds for $n > k$. Consider the $(n+1)$-th packet. The probability of a k-size subset that does not include the $(n+1)$-th packet is the probability that the k-size subset was selected after reading the n-th packet and the $(n+1)$-th packet was not selected. These two events are independent, which means the probability of selecting such a subset is

$$\frac{1}{\binom{n}{k}}\left(1 - \frac{k}{n+1}\right) = \frac{k!(n-k)!}{n!}\left(\frac{n+1-k}{n+1}\right) = \frac{k!(n+1-k)!}{(n+1)!}.$$

This simplifies to $\frac{1}{\binom{n+1}{k}}$, so induction goes for subsets excluding the $n+1$ element.

The probability of a k-size subset H that includes the $(n+1)$-th packet p_{n+1} can be computed as follows. Let G be a k-size subset of the first n packets. The only way we can get from G to H is if G contains $H \setminus \{p_{n+1}\}$. Let G^* be such a subset; let $\{q\} = G \setminus G^*$.

The probability of going from G to H is the probability of selecting p_{n+1} and dropping q, which is equal to $\frac{k}{n+1}\frac{1}{k}$. There exist $-(k-1)$ candidate subsets for G^*, each with probability $\frac{1}{\binom{n}{k}}$ (by the inductive hypothesis) which means that the probability of H is given by

$$\frac{k}{n+1}\frac{1}{k}(n+(k-1))\frac{1}{\binom{n}{k}} = \frac{(n+1-k)(n-k)!k!}{(n+1)n!} = \frac{1}{\binom{n+1}{k}},$$

so induction goes through for subsets including the $(n+1)$-th element. Following is the code in C++:

```
template <typename T>
vector<T> reservoir_sampling(istringstream &sin, const int &k) {
  T x;
  vector<T> R;
  // Store the first k elements
  for (int i = 0; i < k && sin >> x; ++i) {
    R.emplace_back(x);
  }

  // After the first k elements
  int element_num = k + 1;
  while (sin >> x) {
    default_random_engine gen((random_device())());  // random num generator
    // Generate random int in [0, element_num]
    uniform_int_distribution<int> dis(0, element_num++);
    int tar = dis(gen);
    if (tar < k) {
      R[tar] = x;
    }
  }
  return R;
}
```

Problem 20.7, *pg. 157*: *Design an algorithm that computes an array of size k consisting of distinct integers in the set $\{0, 1, \ldots, n-1\}$. All subsets should be equally likely and, in addition, all permutations of elements of the array should be equally likely. Your time should be $O(k)$. Your algorithm should use $O(k)$ space in addition to the k element array holding the result. You may assume the existence of a subroutine that returns integers in the set $\{0, 1, \ldots, n-1\}$ with uniform probability.*

Solution 20.7: We maintain a hash table H which maps a subset of $\{0, 1, \ldots, n-1\}$ to $\{0, 1, \ldots, k-1\}$. Initially H is empty. The final result is stored in an array R of length k. We do k iterations of the following. Choose a random integer r in $[i, n-1]$, where i is the current iteration count, starting at 0. If r does not lie in H, we set $R[i]$ to r and

add (r, i) to H. If r does lie in H, say $(r, j) \in H$, we set $R[i] = j$, remove (r, j) from H, and add (r, i) to H.

This approach is correct because it mimics the offline sampling algorithm described in Solution 20.2 on Page 427. We simulate the array used in that algorithm with H; when $k \ll n$, this results in a huge saving in space, since the array is of length n, and most of it is unchanged.

```
vector<int> online_sampling(const int &n, const int &k) {
  unordered_map<int, int> table;
  vector<int> res;
  for (int i = 0; i < k; ++i) {
    default_random_engine gen((random_device())());  // random num generator
    // Generate random int in [i, n - 1]
    uniform_int_distribution<int> dis(i, n - 1);
    int r = dis(gen);
    auto it = table.find(r);
    if (it == table.end()) {  // r is not in table
      res.emplace_back(r);
      table.emplace(r, i);
    } else {  // r is in table
      res.emplace_back(it->second);
      it->second = i;
    }
  }
  return res;
}
```

Problem 20.8, *pg. 157*: *Assuming elections are statistically independent and that the probability of a Republican winning Election i is p_i, how would you compute the probability of a Republican majority?*

Solution 20.8: Number the individual elections from 1 to 435. Let p_n be the probability that the Republican candidate wins Election n. Let $\Pr(r, n)$ be the probability that exactly r Republicans win in elections $\{1, 2, \ldots, n\}$.

Exactly r Republicans win in elections $\{1, 2, \ldots, n\}$ if (1.) r Republicans win in elections $\{1, 2, \ldots, n-1\}$ and the Republican candidate loses election n, or (2.) $r-1$ Republicans win in elections $\{1, 2, \ldots, n-1\}$ and the Republican candidate wins election n.

Since these events are disjoint, $\Pr(r, n)$ is the sum of the probabilities of these two events. To be precise,

$$\Pr(r, n) = \Pr(r-1, n-1)p_n + \Pr(r-1, n)(1 - p_n).$$

Therefore P can be computed using DP; the base cases for the recursion are $\Pr(0, 0) = 1$ and $\Pr(r, n) = 0$, for $r > n$.

The probability of a Republican majority is $\sum_{k=\lceil \frac{435}{2} \rceil}^{435} \Pr(k, 435)$. Since both r and n take values from 0 to the total number of elections and computing $\Pr(r, n)$ from

earlier values takes $O(1)$ time, the complexity of computing $\{\Pr(i, j) \mid 0 \le i \le j \le 435\}$ is proportional to the square of the number of elections.

```
// prob is the probability that each Republican wins.
// r is the number of Republicans wins, and n is the number of elections.
double house_majority_helper(const vector<double> &prob, const int &r,
                             const int &n, vector<vector<double>> &P) {
  if (r > n) {
    return 0.0;  // base case: not enough Republicans
  } else if (r == 0 && n == 0) {
    return 1.0;  // base case
  } else if (r < 0) {
    return 0.0;
  }

  if (P[r][n] == -1.0) {
    P[r][n] = house_majority_helper(prob, r - 1, n - 1, P) * prob[n - 1] +
              house_majority_helper(prob, r, n - 1, P) * (1.0 - prob[n - 1]);
  }
  return P[r][n];
}

double house_majority(const vector<double> &prob, const int &n) {
  // Initialize DP table
  vector<vector<double>> P(n + 1, vector<double>(n + 1, -1.0));

  // Accumulate the probabilities of majority cases
  double prob_sum = 0.0;
  for (int r = ceil(0.5 * n); r <= n; ++r) {
    prob_sum += house_majority_helper(prob, r, n, P);
  }
  return prob_sum;
}
```

ϵ-Variant 20.8.1: Compute the probability of a Republican majority given the outcomes of a subset of the races.

ϵ-Variant 20.8.2: Richard and Leopold are playing a series of tennis games in which Richard wins an individual game with probability 0.6. The outcomes of successive games are independent. The first player to win 11 games wins the series. The prize for winning the series is \$100; the winner gets all the prize money. Because of the weather, the series is stopped with Richard leading 7 games to 5. Divide the prize money fairly between Richard and Leopold.

Variant 20.8.3: Consider the following three events: getting one or more sixes when six dice are rolled, getting two or more sixes when 12 dice are rolled, and getting three or more sixes when 18 dice are rolled. Which, if any, of these three events is most probable?

Problem 20.9, *pg. 158*: *You select a coin at random from the bag and toss it five times.*

It comes up heads three times. What is the probability that it was the coin that was biased towards tails? How many times do you need to toss the coin that is biased towards tails before it comes up with a majority of tails with probability greater than $\frac{99}{100}$?

Solution 20.9: Let L be the event that the selected coin is tail-biased, U be the event that the selected coin is head-biased, and $3H5$ be the event that a coin chosen at random from the bag comes up heads 3 times out of 5 tosses.

We want to compute $\Pr(L \mid 3H5)$. By Bayes' rule, this is $\Pr(L \cap 3H5)/\Pr(3H5)$. Applying Bayes' rule again, this probability equals

$$\begin{aligned}
&\frac{\Pr(3H5 \mid L)\Pr(L)}{\Pr\big(3H5 \cap (L \cup U)\big)} \\
&= \frac{\Pr(3H5 \mid L)\Pr(L)}{\Pr(3H5 \cap L) + \Pr(3H5 \cap U)} \\
&= \frac{\Pr(3H5 \mid L)\Pr(L)}{\Pr(3H5 \mid L)\Pr(L) + \Pr(3H5 \mid U)\Pr(U)} \\
&= \frac{\binom{5}{3} \times 0.4^3 \times 0.6^2 \times 0.5}{\binom{5}{3} \times 0.4^3 \times 0.6^2 \times 0.5 + \binom{5}{3} \times 0.4^2 \times 0.6^3 \times 0.5} \\
&= 0.4
\end{aligned}$$

For the second part, we can use the Chebyshev inequality to compute the number of trials we need for a majority of n tosses of the tail-biased coin to be heads with probability $\frac{1}{100}$. Let L_i be the event that the i-th toss of the tail-biased coin comes up heads. It will be convenient to use a Bernoulli random variable X_i to encode this event, with a 1 indicating heads and 0 indicating tails.

The mean μ of the sum X of n Bernoulli random variables which are independent and identically distributed (IID) with probability p is $n \times p$; the standard deviation σ is $\sqrt{np(1-p)}$. In our context, $\mu = 0.4n$ and $\sigma = \sqrt{6n/25}$.

The Chebyshev inequality gives us an upper bound on the probability of a random variable being far from its mean. Specifically, $\Pr\big(|X - \mu| \geq k\sigma\big) \leq \frac{1}{k^2}$. Note that the event $|X - \mu| \geq k\sigma$ is a superset of the event $X - \mu \geq k\sigma$.

For the majority of n tosses to not be tails, it is necessary that the sum of the n coin tosses is greater than or equal to $0.5n$. To bound this probability by $\frac{1}{100}$ we take $k = 10$ in the Chebyshev inequality. Specifically, we need to solve for n such that $0.5n - 0.4n \geq 10 \times \sqrt{6n/25}$, i.e., $0.1n \geq 10 \times \sqrt{6n/25}$, which is satisfied for $n \geq 2400$.

The Chebyshev inequality holds for all random variables if they have a variance. We can obtain a tighter bound by applying a Chernoff bound, which is specific to the sums of Bernoulli random variables. Specifically, Chernoff bounds tell us that $\Pr(X \geq (1+\delta)\mu) \leq e^{\frac{-\mu\delta^2}{3}}$. We want to bound $\Pr(X \geq 0.5n = (1 + 0.25)(0.4n))$, hence $\delta = 0.25$. Thus we want $e^{\frac{-0.4n(0.25)^2}{3}} < 0.01$; taking natural logs we obtain $-\frac{0.4n(0.25)^2}{3} < -\ln 100 = -4.6$, which holds for $n > 552$.

The Chernoff bound is also pessimistic. Through a simulation, the code for which is attached below, we determined that when $n = 553$, only 17 times in 10^7 trials did

we observe a majority of tails. When $n = 148$, tails was not a majority in 0.88% of the trials.

```
// Return the number of fail trails
int simulate_biased_coin(const int &n, const int &trails) {
  default_random_engine gen((random_device())());  // random num generator
  // Generate random double in [0.0, 1.0]
  uniform_real_distribution<double> dis(0.0, 1.0);
  const double bias = 0.4;
  int fails = 0;
  for (int i = 0; i < trails; ++i) {
    int biased_num = 0;
    for (int j = 0; j < n; ++j) {
      biased_num += (dis(gen) >= bias);
    }

    if (biased_num < (n >> 1)) {
      ++fails;
    }
  }
  return fails;
}
```

Problem 20.10, *pg. 158*: *If m balls are thrown into n bins uniformly randomly and independently, what is the expected number of bins that do not have any balls?*

Solution 20.10: The probability that a given ball does not land in a given bin is $\frac{n-1}{n}$. Since throws are independent, the probability that no ball lands in that bin is $\left(\frac{n-1}{n}\right)^m$. Hence the expected number of empty bins is $n\left(\frac{n-1}{n}\right)^m$. This is closely approximated by $n \times e^{-m/n}$. Hence if on an average, each server is handling significantly more than one client, there should be very few idle servers. If $n = m$, then the expected number of empty bins tends to $1/e$ times the total number of bins, which is a classical result.

We used the linearity of expectation in an essential way. Linearity of expectation does not require the individual random variables to be independent. This is crucial, since bins are not independent, e.g., it is impossible for all bins to be empty.

Variant 20.10.1: David is the first passenger to board a flight. He has lost his boarding card, and selects a seat to sit in uniformly randomly. Successive passengers either sit in their assigned seat, or, if someone is already in their seat, select another seat uniformly randomly from the set of remaining empty seats. Selections are done independently. Henri is the first to board a different flight. He has also lost his boarding card, and his flight fills up the same way as David's flight. There are 100 seats on David's flight, and 200 seats on Henri's flight. Both flights are full. Let L_A and L_B be the last passengers to board David's flight and Henri's flight, respectively. Which of L_A and L_B are more likely to get their assigned seat?

Variant 20.10.2: How many people need to be at a party before the probability of two people at the party having a common birthday exceeds 0.5? How many people

need to be at a party before the probability of one of them having your birthday exceeds 0.5? (Assume birthdays are uniformly independently distributed across 365 days of the year; nobody is born on February 29.)

Problem 20.11, *pg. 158*: *What is the expected number of fixed points of a uniformly random permutation $\sigma : \{0, 1, \ldots, n-1\} \mapsto \{0, 1, \ldots, n-1\}$, i.e., the expected cardinality of $\{i \mid \sigma(i) = i\}$? What is the expected length of the longest increasing sequence starting at $\sigma(0)$, i.e., if k is the first index such that $\sigma(k) < \sigma(k-1)$, what is the expected value of k?*

Solution 20.11: Let X_i be the random variable, which is 1 if $\sigma(i) = i$ and 0 otherwise. Such a random variable is often referred to as an "indicator random variable". The number of fixed points is equal to $X_0 + X_1 + \cdots + X_{n-1}$. Expectation is linear, i.e., the expected value of a sum of random variables is equal to the sum of the expected values of the individual random variables. The expected value of X_i is $0 \times \frac{n-1}{n} + 1 \times \frac{1}{n}$ (since an element is equally likely to be mapped to any other element). Therefore the expected number of fixed points is $n \times \frac{1}{n} = 1$.

We can compute the expected value of k by defining indicator random variables $Y_0, Y_1, \ldots Y_{n-1}$, where $Y_i = 1$ iff for all $j < i$ we have $\sigma(j) < \sigma(i)$. Observe that k is simply the sum of the Y_is. The expected value of Y_i is $\frac{1}{i+1}$, since for all $j < i$ we have $\sigma(j) < \sigma(i)$ iff the largest of the first $i+1$ elements is at position i, which has probability $\frac{1}{i+1}$ since all the permutations are equally likely. Therefore the expected value for k is $1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}$, which tends to $\log_e n$.

For both parts of the problem, we used the linearity of expectation which does not require the individual random variables to be independent. This property of expectation is crucial since the X_is and the Y_js are not independent—for example, if the first $n-1$ elements get mapped to themselves, then the n-th element must also map to itself.

Problem 20.12, *pg. 158*: *Gottfried repeatedly rolls an unbiased six-sided die. He stops when he has rolled all the six numbers on the die. How many rolls will it take, on an average, for Gottfried to see all the six numbers?*

Solution 20.12: First we prove that if $\langle X_0, X_1, \ldots \rangle$ is a sequence of Bernoulli IID random variables, with $p(X_i = 1) = p$, then the expected time to see the first 1 is $\frac{1}{p}$. The reasoning is as follows. Define F_i to be the event that the first 1 comes on the i-th trial. Then $\Pr(F_i) = (1-p)^{i-1}p$. Hence the expected time is $S = \sum_{i=1}^{\infty} i(1-p)^{i-1}p$. This sum simplifies to $\frac{1}{p}$ (multiply both sides by p, subtract, and sum the infinite geometric series on the right).

Now, we consider the problem of die rolls. The key is to determine the expected time to see the k-th new value. Clearly, the expected time to see the first new value is just 1. The time to see the second new value from the first new value is $\frac{1}{5/6}$ since the probability of seeing a new value, given that one value has already been seen, is $\frac{5}{6}$. In this way, the time taken to see the third new value, given that two values have already been seen, is $\frac{1}{4/6}$. Generalizing this idea, the time taken to see the k-th new value, given that $k-1$ values have already been seen, is $\frac{1}{(6-(k-1))/6}$. Hence the expected

time to see the sixth new value is $\frac{6}{6} + \frac{6}{5} + \frac{6}{4} + \frac{6}{3} + \frac{6}{2} + \frac{6}{1} \approx 14.7$.

Variant 20.12.1: On average, how many cards on average have to be dealt from a well-shuffled deck before an ace appears?

Problem 20.13, *pg. 159*: *What is the probability that these three segments can be assembled into a triangle?*

Solution 20.13: The first thing to note is that three segments can make a triangle iff no one segment is longer than the sum of the other two. The "only if" follows from the triangle inequality and the "if" follows from a construction—take a segment and draw circles at the endpoints with radius equal to the lengths of the other circles.

For the case we are considering the three segment lengths add up to 1. Therefore, there exists a segment that is longer than the sum of the other two iff there exists a segment that is longer than $\frac{1}{2}$.

Let $l = \min(u1, u2)$, $m = \max(u1, u2) - \min(u1, u2)$, and $u = 1 - \max(u1, u2)$; these are the lengths of the first, second, and third segments, from left-to-right. If one segment is longer than 0.5, then none of the others can be longer than 0.5, implying the events $l > 0.5$, $m > 0.5$, and $u > 0.5$ are disjoint.

Observe that $l > 0.5$ iff both $u1$ and $u2$ are greater than 0.5; the probability of this event is $\frac{1}{2} \times \frac{1}{2}$ because $u1$ and $u2$ are chosen independently. Similarly $u > 0.5$ iff both $u1$ and $u2$ are less than 0.5, which has probability $\frac{1}{2} \times \frac{1}{2}$.

To compute the probability of $m > 0.5$, first we consider the case that $u1 < u2$. For $m > 0.5$, we need $u1$ to be in $[0, 0.5]$ and $u2$ to be in $[0.5 + u1, 1]$. This probability can be expressed by the integral

$$\int_{u1=0}^{0.5} \int_{u2=u1+0.5}^{1} 1 \times du1 \times du2$$

which evaluates to $\frac{1}{8}$.

By symmetry, the probability of $m > 0.5$ when $u1 > u2$ is also $\frac{1}{8}$. Hence the probability of a segment being longer than $\frac{1}{2}$ is $\frac{1}{4} + \frac{1}{4} + \frac{1}{4} = \frac{3}{4}$. Therefore the probability of being able to make a triangle out of the segments is $1 - \frac{3}{4} = \frac{1}{4}$.

ϵ-Variant 20.13.1: If a stick is broken into two pieces uniformly randomly, what is the expected length of the longer piece? What is the expected value of the ratio of the length of the longer piece to the length of the smaller piece?

ϵ-Variant 20.13.2: If a stick is broken into three pieces uniformly randomly, what is the expected length of the longest, shortest and middle-sized pieces?

Problem 20.14, *pg. 159*: *Solve Problem 20.13 on Page 159 when $u1$ is uniformly randomly in $[0, 1]$ and $u2$ is subsequently chosen uniformly randomly in $[u1, 1]$. Can you determine which of these two approaches is more likely to produce a triangle without computing the exact probabilities?*

Solution 20.14: We fail to make a triangle in case $u1 > 0.5$, $u2 - u1 > 0.5$, or $1 - u2 > 0.5$. The first probability is simply $\frac{1}{2}$.

The second probability is given by the integral

$$\int_{u1=0}^{0.5} \int_{u2=u1+0.5}^{1} \frac{1}{(1-u1)} \times du2 \times du1.$$

Note that the probability density function for $u2$ is different from the previous case since $u2$ is uniform in $[u1, 1]$, not $[0, 1]$. This integral evaluates to $\frac{1+\log_e \frac{1}{2}}{2}$. The third probability can also be computed using an integral but by symmetry, it must be the same as the second probability. Hence the final probability is $\frac{1}{2} + 2 \times \frac{1+\log_e \frac{1}{2}}{2} \approx 0.807$.

Intuitively, the second formulation leads to a higher probability of a long line segment (which implies that we cannot form a triangle) because there is less diversity in the points. For the first case, the points are spread randomly; for the second, there is a 0.5 probability that the first point itself precludes us from building the triangle. Another way to think of it is that if we put down many points, the first method will lead to short segments with little variation in lengths but the second method will give us a skewed distribution and the first few segments will be considerably longer.

Problem 20.15, *pg. 159*: *Design a strategy that selects the best secretary with a probability greater than* 0.25, *regardless of* n.

Solution 20.15: The only reasonable interpretation of random is that all permutations are equally likely. Therefore, if we always select the first secretary, we have a $\frac{1}{n}$ chance of selecting the best secretary.

One way to do better is to skip the first $\lfloor \frac{n}{2} \rfloor$ secretaries and then choose the first one in the remaining set that is superior to the best secretary interviewed in the first $\lfloor \frac{n}{2} \rfloor$ secretaries. Roughly, the probability of selecting the best secretary with this strategy is at least $\frac{1}{4}$ since the probability that the second best secretary lies in the first set and the best secretary is in the second set is at least $\frac{1}{4}$.

The probability is actually greater than $\frac{1}{4}$ since if the second best secretary is in the first set, there is a higher than 0.5 probability that the best secretary is in the second set. When n is even, say $n = 2m$, the probability that the strategy selects the best secretary is at least $\frac{m}{2m} \frac{m+1}{2m} > \frac{1}{4}$. When n is odd, say $n = 2m+1$, the probability that the strategy selects the best secretary is at least $\frac{m}{2m+1} \frac{m+1}{2m} > \frac{1}{4}$. Note that even if the second best secretary is not in the first set, our strategy may still select the best secretary, e.g., if the best secretary is the first to appear.

It is known that if we follow a strategy of skipping the first s secretaries and selecting the first secretary who is superior to all others so far, the probability of selecting the best secretary is maximized for s closest to $\frac{n}{e}$, and this probability tends to $\frac{1}{e}$.

Problem 20.16, *pg. 159*: *What is the value of* w *such that Once-or-Twice is a fair game, i.e., for a rational player, the expected gain is* 0*?*

Solution 20.16: If the probability of winning is p, then the expected gain is $-1+p\times w$. Hence for a fair game, $w=\frac{1}{p}$.

The face value of the card can be any number between 1 and 13. For the dealer, all values are equally likely. Hence if the player's card has a face value i, then the probability of winning for the player is $\frac{i-1}{13}$. If the player always takes only one random card, his probability of winning is $\frac{1}{13}\sum_{i=1}^{13}\frac{i-1}{13}=\frac{6}{13}$. Hence it makes sense to ask for the next card only if the first card yields a probability less than $\frac{6}{13}$, i.e., the face value of the first card is less than 7. If the face value of the first card is 7 or more, then the probability of winning is $\frac{1}{7}\sum_{i=7}^{13}\frac{i-1}{13}=\frac{9}{13}$; otherwise, we pick again, in which case the probability of winning is $\frac{6}{13}$. Hence the overall probability of winning is $\frac{7}{13}\frac{9}{13}+\frac{6}{13}\frac{6}{13}=\frac{99}{169}$. Thus the fair value is $\frac{169}{99}\approx 1.707$.

Problem 20.17, *pg. 159*: *Suppose you are playing the multibet card color game and are restricted to bet in penny increments. Compute a tight lower bound on the amount that you can guarantee to win under this restriction.*

Solution 20.17: A good way to begin is to devise a strategy that guarantees a positive return. It is possible to guarantee a 2× return by waiting till the last card and betting the entire amount on the last card whose color is uniquely determined by the 51 cards that have already been seen.

To do better than a 2× return, consider the case of a deck of 4 cards with 2 red cards and 2 black cards. If we do not bet on the first card, there will be three remaining cards. Assume, without loss of generality, that two cards are black and one is red. If we bet $\$\frac{1}{3}$ on the next card being black and are successful, then we have $\$\frac{4}{3}$ which we can double on the last card for a $\frac{8}{3}>2$ return. If we lose, then the two remaining cards are black, in which case we can double our remaining money twice, i.e., achieve a $\frac{2}{3}\times 2\times 2=\frac{8}{3}>2$ return. Note that this analysis assumes we can bet arbitrary fractions of the money we possess.

Now, we consider the case where we can only bet in penny increments. Let $Q(c,r,t)$ be the maximum we can guarantee, when we have c cents to gamble with and there are r red cards remaining out of a total of t cards. We can bet b cents, for $0\le b\le c$ on the next card. Since we have to design a strategy that maximizes the worst-case payoff, the maximum amount we can make on betting on red cards is given by

$$Q_R(c,r,t) \quad = \quad \max_{0\le b\le c}\Big(\min\big(Q(c+b,r-1,t-1),Q(c-b,r,t-1)\big)\Big).$$

The maximum we can make by betting on black cards is

$$Q_B(c,r,t) \quad = \quad \max_{0\le b\le c}\Big(\min\big(Q(c+b,r,t-1),Q(c-b,r-1,t-1)\big)\Big).$$

Hence $Q(c,r,t)=\max\big(Q_R(c,r.t),Q_B(c,r,t)\big)$ which yields a DP algorithm for computing the maximum payoff—base cases are of the form $Q(c,0,t)$ and $Q(c,t,t)$, both of which are $c\times 2^t$.

However if we directly try and compute $Q(100, 26, 52)$, the algorithm runs for an unacceptably long time. This is because we will be exploring paths for which c grows very large. Since we are given the maximum payoff on a dollar when fractional amounts can be bet is less than 9.09, we can prune computations for $Q(c, r, t)$ when $c \geq 909$. The following code implements the DP algorithm with this pruning; it computes the maximum payoff, 808, in two minutes.

```
double compute_best_payoff_helper(
    unordered_map<int,
                  unordered_map<int, unordered_map<int, double>>> &cache,
    const double &upper_bound, const int &cash, const int &num_red,
    const int &num_cards) {
  if (cash >= upper_bound) {
    return cash;
  }

  if (num_red == num_cards || num_red == 0) {
    return cash * pow(2, num_cards);
  }

  if (cache[cash][num_red].find(num_cards) == cache[cash][num_red].end()) {
    double best = numeric_limits<double>::min();
    for (int bet = 0; bet <= cash; ++bet) {
      double red_lower_bound = min(
        compute_best_payoff_helper(cache, upper_bound, cash + bet,
                                   num_red - 1, num_cards - 1),
        compute_best_payoff_helper(cache, upper_bound, cash - bet,
                                   num_red, num_cards - 1));

      double black_lower_bound = min(
        compute_best_payoff_helper(cache, upper_bound, cash - bet,
                                   num_red - 1, num_cards - 1),
        compute_best_payoff_helper(cache, upper_bound, cash + bet,
                                   num_red, num_cards - 1));
      best = max(best, max(red_lower_bound, black_lower_bound));
    }
    cache[cash][num_red][num_cards] = best;
  }
  return cache[cash][num_red][num_cards];
}

double compute_best_payoff(const int &cash) {
  double upper_bound = 9.09 * cash;
  unordered_map<int, unordered_map<int, unordered_map<int, double>>> cache;
  return compute_best_payoff_helper(cache, upper_bound, cash, 26, 52);
}
```

Here is a sketch of the proof that $2^{52}/\binom{52}{26}$ is the maximum amount that you can guarantee you will win when arbitrary fractions of the stake can be bet.

Let $M(r, b)$ be the maximum amount you can guarantee you will win starting with \$1 when there are r red cards and b black cards remaining, under the assumption that you can bet any fraction of your current stake. Let $f(r, b) \in [-1, 1]$ be the optimum amount to bet on red; $f(r, b)$ could be negative, which is equivalent to betting a

positive amount on black. Then M and f satisfy the following:

$$M(r,b) = \max_{f(r,b)\in[-1,1]} (\min((1+f(r,b))M(r-1,b), (1-f(r,b))M(r,b-1))).$$

Elementary algebra can be used to show that the value for $f(r,b)$ that maximizes $M(r,b)$ is the one in which $(1+f(r,b))M(r-1,b) = (1-f(r,b))M(r,b-1)$; both of these terms equal $M(r,b)$. The optimum value for $f(r,b)$ is $\frac{r-b}{r+b}$. Substituting for $f(r,b)$ in leads to the following recurrence:

$$M(r,b) = \frac{2M(r-1,b)M(r,b-1)}{M(r-1,b)+M(r,b-1)}.$$

The base cases are $M(r,0)$ and $M(0,b)$, which are 2^r and 2^b, respectively. Induction can be used to prove that $M(r,b) = 2^{r+b}/\binom{r+b}{b}$. Substituting $r = b = 26$ yields the desired result.

Problem 20.18, *pg. 160*: *Design a strategy that maximizes the probability of winning at the one red card game.*

Solution 20.18: We can trivially achieve a probability of success of $\frac{1}{2}$ by always choosing the first card.

A natural way to proceed is to consider the probability $p_k(f)$ of winning for the optimum strategy after k cards remain, of which f are red cards. Then $p_k(f) = \max\left(\frac{f}{k}, \frac{f}{k}p_{k-1}(f-1) + (1-\frac{f}{k})p_{k-1}(f)\right)$.

The base cases for the recurrence are $p_1(1) = 1$ and $p_1(0) = 0$. Applying the recurrence, we obtain $p_2(2) = 1, p_2(1) = \frac{1}{2}, p_2(0) = 0$, and $p_3(3) = 1, p_3(2) = \frac{2}{3}, p_3(1) = \frac{1}{3}, p_3(0) = 0$. This suggests that $p_k(f) = \frac{f}{k}$, which can directly be verified from the recurrence. Therefore the best we can do, $p_{52}(26) = \frac{26}{52} = \frac{1}{2}$, is no better than simply selecting the first card.

An alternate view of this is that since the cards in the deck are randomly ordered, the probability of the topmost card being red is the same as that of the card at the bottom of the deck being red. The bottom-most card has a $\frac{f}{k}$ probability of being red when there are f red cards and k cards in total.

Variant 20.18.1: An alchemist has discovered several dryads—stones which can change lead into gold. The rate at which a dryad converts lead into gold is proportional to its weight. The alchemist also has tryads, stones which, by themselves, do nothing. However, when a tryad is heated together with a dryad, the two stones combine to form a new stone, which could be either a tryad or a dryad. The weight is conserved. The probability of the combined stone being a dryad is the weight of the initial dryad divided by the sum of the weights of the initial stones. He can also combine tryads with tryads and dryads with dryads, which results in the same stone with the sum of the constituent weights.

The alchemist wants to combine tryads with dryads to maximize the expected dryad weight. He could follow many strategies, such as combining the heaviest

dryad with the lightest tryad, or combining the dryad and tryad that are closest in weight. What strategy should he follow?

Variant 20.18.2: Isaac and Leonhard are playing a card game against each other. Isaac starts with \$2 and Leonhard starts with \$1. They bet \$1 on each game. They stop playing when one of the two runs out of money; the other is the overall winner. Suppose the probability of Leonhard winning any single game is $\frac{2}{3}$. What is the probability that he is the overall winner?

Problem 20.19, *pg. 160*: *Consider an auction for an item in which the reserve price is a random variable X uniformly distributed in $[0, 400]$. You can bid B. If your bid is greater than or equal to the reserve price, you win the auction and have to pay B. You can then sell the item for an 80% markup over the reserve price. How much should you offer for the item?*

Solution 20.19: The first question to ask is what are you trying to optimize? The objective could be to maximize expected profit, minimize loss, or maximize ratio of expected profit to variance.

Let's say we want to maximize expected profit. Let X be the random variable corresponding to the reserve price. We win the auction if the reserve price is less than or equal to our bid. The selling price is $1.8\times$ the reserve price, and our cost to buy is fixed at B. Therefore, the expected profit is $\int_{X=0}^{X=B} \frac{1}{400}(1.8X - B)dX$. This simplifies to $\frac{0.9\times B^2 - B^2}{400}$, which is negative for all $B > 0$, i.e., we should not place a bid.

In retrospect, this result is obvious since if we win the auction, we are paying twice of X in expectation and getting only $1.8X$ in return.

Problem 20.20, *pg. 160*: *Your friend at the Acme Casino has rigged their roulette wheel to make the probability of the ball landing on red $\frac{19}{37}$. You can bet on the same color exactly 100 times; after that the casino management will be alerted. You start with \$1. On each round, you can bet any amount from 0 to your entire bankroll. What should your strategy be?*

Solution 20.20: The key to solving this problem is determining what the objective is. The "obvious" criterion is expectation maximization. It is simple to see that the strategy that maximizes expectation is betting the entire bankroll on red each time. The expected payoff is $2^{100} \times \frac{19}{37}^{100} \approx 3\times 10^{16}$; however, the probability that the strategy does not result in a "bust" (the bankroll going to 0) is $\frac{19}{37}^{100} \approx 2.39 \times 10^{-14}$.

We can avoid busting by maximizing the expectation of the logarithm of the final bankroll—since $\log 0 = -\infty$, any strategy which maximizes this objective will never bet the whole amount on any one bet. Let p be the odds of winning, and $q = 1 - p$ the odds of losing. If we bet r fraction of the current bankroll B, the expectation of the logarithm of the resulting bankroll is $p \log(B + 2rB) + (1 - p)\log(B - rB)$. Simple calculus shows that the optimum choice of $r = p - q = 2p - 1$.

The above analysis was first done by John Larry Kelly, a Ph.D. from The University of Texas at Austin, who went on to work at Bell Labs. The best reference is his original paper, *"A New Interpretation of Information Rate"*. William Poundstone's *"Fortune's*

Formula: The Untold Story of the Scientific Betting System That Beat the Casinos and Wall Street" is a highly readable account of Kelly's result and its impact

Variant 20.20.1: Find the optimum betting strategy when there are k outcomes, each with probability p_i and return r_i.

Variant 20.20.2: You can choose among k investment types. The return on the i-th investment type has a normal distribution with mean r_i and standard deviation σ_i^2. The covariance between investment types i and j is σ_{ij}. Compute a portfolio, i.e., an allocation of an investment across these k investment types that maximizes the expected return, subject to the constraint that the aggregate investment has a variance that is less than a specified constant. How does your approach handle short positions, i.e., selling an investment type that you do not own? How would you handle diversification constraints, e.g., a requirement of the form "no more than 20% of the portfolio can be in any four investment types"? How would you do the same if there is uncertainty in some of the parameters?

Variant 20.20.3: Design a strategy by which you can go to a casino with \$100, play an unrigged Roulette wheel, and leave with more than \$100 with probability greater than 0.98. The casino only accepts positive integer-valued bets.

Variant 20.20.4: Suppose you can bet a dollar on the ball falling into a pocked numbered 1 to 36. If the ball falls in the pocket you selected, you receive \$36 in return; otherwise, you lose your bet. The casino offers you insurance against losing. Specifically, for \$20 you can insure 36 one dollar bets. If you are behind after 36 bets, the casino will give you \$40, otherwise it keeps the premium. Should you buy the insurance?

Problem 20.21, *pg. 160*: *Prove that an algorithm in which the choice of the next variable to read in an L_K expression is a deterministic function of the values read up to that point must, in the worst case, read all variables to evaluate the expression. Design a randomized algorithm that reads fewer variables on an average, independent of the values assigned to the variables.*

Solution 20.21: First, we show that any deterministic algorithm must examine all the Boolean variables. The idea is that an adversary can force the value of any subexpression to be unknown till all the variables in that subexpression have been read. For example, suppose variable X is ANDed with variable Y. If the algorithm reads the value of X before Y, we return true; when Y is queried, we return false. In this way, the value of $X \wedge Y$ is determined only after both the variables are read.

This generalizes with induction: the inductive hypothesis is that an L_k expression requires all the variables to have been read before its value is determined and its final value is the value of the last variable read. For a subexpression of the form $\phi \wedge \psi$, where ϕ and ψ are L_k expressions, if all the variables from ϕ are read before

all the variables from ψ are read, the adversary chooses the last variable read from ϕ to be true, forcing the algorithm to evaluate ψ. A similar argument can be used for subexpressions of the form $\phi \vee \psi$.

Suppose we evaluate an expression by choosing one of its two subexpressions at random to evaluate first; we evaluate the other subexpression only if the expression's value is not forced by the subexpression that we evaluated first.

For example, if we are to evaluate an L_{k+1} expression of the form $((\phi_0 \wedge \phi_1) \vee (\psi_0 \wedge \psi_1))$, where the subexpressions $\phi_0, \phi_1, \psi_0, \psi_1$ are L_k expressions, we randomly choose one of $(\phi_0 \wedge \phi_1)$ and $(\psi_0 \wedge \psi_1)$ to evaluate first. If the first expression evaluated is true, we can ignore the second; otherwise, we evaluate the second. If the first expression is true, we reduced the number of variables queried by at least half. If the first expression is false, at least one of the two subexpressions is false and we have a probability of 0.5 of selecting that subexpression and avoiding evaluating the other subexpression. So, in the worst-case, we can expect to avoid one of the four subexpressions $\phi_0, \phi_1, \psi_0, \psi_1$. Therefore the expected number of variables queried to evaluate an L_{k+1} expression, $Q(k+1)$ satisfies

$$Q(k+1) \leq 3Q(k).$$

From this, $Q(k) = 3^k$. It is straightforward to use induction to show that an L_k expression contains $n = 4^k$ variables, so $Q(k) = n^{\log_4 3} = n^{0.793}$.

Problem 20.22, *pg. 161*: *For what option price is there no opportunity for arbitrage?*

Solution 20.22: Let f be the price for the option. A fair price is determined by the no-arbitrage requirement. Suppose we start with a portfolio of x shares and y options in S—x and y may be negative (which indicates that we sell stocks or sell options).

The initial value of our portfolio is $100x + yf$. On Day 100, two things may have happened:

- The stock went up and the portfolio is worth $120x + 20y$.
- The stock went down and the portfolio is worth $70x$.

If we could choose x and y in such a way that our initial portfolio has a negative value—which means that we are paid to take it on—and regardless of the movement in the stock, our portfolio takes a nonnegative value, then we will have created an arbitrage.

Therefore the conditions for an arbitrage to exist are:

$$\begin{aligned} 120x + 20y &\geq 0 \\ 70x &\geq 0 \\ 100x + yf &< 0 \end{aligned}$$

A fair price for the option is one in which no arbitrage exists. If f is less than 0, an arbitrage exists—we are paid to buy options, lose nothing if the price goes down, and make \$20 per option if the price goes up. Therefore $f \geq 0$, so we can write the third

inequality as $y < -\frac{100}{f}x$. The first equation can be rewritten as $y \geq -6x$. Combining these two inequalities, we see that an arbitrage exists if $-\frac{100}{f} < -6$, i.e., $f > \frac{100}{6}$.

In summary, there is no arbitrage for $f \in [0, \frac{100}{6}]$; there is an arbitrage for all other values of f.

For example, if $f = 19 > \frac{100}{6}$, then the option is overpriced and we should sell ("write") options. If we write b options and buy one share, we will start with a portfolio that is worth $100 + 19b$. If the stock goes down, the options are worthless and our portfolio is worth \$70. If the stock goes up, we lose \$20 on each option we wrote but see a gain on the stock we bought. We want the net gain to be nonnegative and the initial portfolio to have a negative value, i.e.,

$$\begin{aligned} 120 + 20b &\geq 0 \\ 100 + 19b &< 0 \end{aligned}$$

Combining the two inequalities, we see that any value of b in $[-6, -\frac{100}{19})$ leads to an arbitrage.

Problem 20.23, *pg. 161*: *Consider the same problem as Problem 20.22 on Page 161, with the existence of a third asset class, namely a bond. A \$1 bond pays \$1.02 in 100 days. You can borrow money at this rate or lend it at this rate. Show there is a unique arbitrage-free price for the option and compute this price.*

Solution 20.23: Suppose our initial portfolio consists of x_0 stocks, x_1 options, and x_2 bonds.

Proceeding as above, we see the condition for an arbitrage to exist is:

$$\begin{aligned} 100x_0 + fx_1 + x_2 &< 0 \\ 120x_0 + 20x_1 + 1.02x_2 &\geq 0 \\ 70x_0 + 1.02x_2 &\geq 0 \end{aligned}$$

Writing the linear terms as Ax, if $det(A) \neq 0$, then there always exists an arbitrage since we can solve $Ax = b$, and when $b^T = (-1, 1, 1)$ there is an arbitrage.

The determinant of A equals $70(1.02f - 20) + 1.02(100 \times 20 - 120f)$. This equals 0 when $f = 640/51 \approx 12.549 = f^*$, so an arbitrage definitely exists if the option price is not equal to f^*.

Conversely, if the option is priced at f^*, $det(A) = 0$ and in particular $A_0 = 0.6275A_1 + 0.3583A_2$, where A_i denotes the i-th row of A. Since A_0 is a linear combination of A_1 and A_2 with positive weights, if $A_1x \geq 0$ and $A_2x \geq 0$, then $A_0x \geq 0$, so no arbitrage can exist.

Problem 20.24, *pg. 162*: *Suppose the price of Jingle stock 100 days in the future is a normal random variable with mean \$300 and standard deviation \$20. What would be the fair price of an option to buy a single share of Jingle at \$300 in 100 days? (Ignore the effect of interest rates.)*

Solution 20.24: Let x be the price of the stock on Day 100. The option is worthless if $x < 300$. If the price is $x \geq 300$, the option is worth $x - 300$ dollars. The expected value of x is given by the integral

$$\int_{300}^{\infty} (x - 300) \times \frac{e^{-\frac{(x-300)^2}{2\times(20)^2}}}{\sqrt{2\pi(20)^2}} dx.$$

The integral can be evaluated in closed form—let $y = x - 300$ and let's write σ instead of 20. The expression above simplifies to

$$\int_{0}^{\infty} y \times \frac{e^{-\frac{y^2}{2\times\sigma^2}}}{\sqrt{2\pi\sigma^2}} dy.$$

The indefinite integral $\int w \times e^{-w^2} dw$ has the closed form solution $-\frac{e^{-w^2}}{2}$, which implies the definite integral equals $\sigma\sqrt{\frac{1}{2\pi}} \approx 0.39\sigma$. Therefore the expected payoff on the option on Day 100 is $0.39 \times 20 = \$7.8$.

Problem 21.1, *pg. 163*: *Which of the* 500 *doors are open after the* 500*-th person has walked through?*

Solution 21.1: As described on Page 38, analyzing a few small examples suggests that, independent of n, door k will be open iff k is a perfect square. This can be rigorously proved as follows.

Proof:

If the number of times a door's state changes is odd, it will be open; otherwise it is closed. Therefore the number of times door k's state changes equals the number of divisors of k. From the small example analysis, we are led to the conjecture that the number of divisors of k is odd iff k is a perfect square. Note that if d divides k, then k/d also divides k. Therefore we can uniquely pair off divisors of k, other than $\sqrt{k}$ (if it is an integer). Hence, when $\sqrt{k}$ is not an integer, k has an even number of divisors. When $\sqrt{k}$ is an integer, it is the only divisor of k that cannot be uniquely paired off with another divisor, implying k has an odd number of divisors. By definition, $\sqrt{k}$ is an integer iff k is a perfect square, proving the result.

This check can be performed by squaring $\lfloor \sqrt{i} \rfloor$ and comparing the result with i.

```
bool is_door_open(const int &i) {
  double sqrt_i = sqrt(i);
  int floor_sqrt_i = floor(sqrt_i);
  return floor_sqrt_i * floor_sqrt_i == i;
}
```

Variant 21.1.1: There are 25 people seated at a round table. Each person has two cards. Each card has a number from 1 to 25. Each number appears on exactly two cards. Each person passes the card with the smaller number to the person on his left.

This is done iteratively in a synchronized fashion. Show that eventually someone will have two cards with identical numbers.

Problem 21.2, *pg. 163*: *What is the minimum number of five man time-trials needed to determine the top three cyclists from a set of 25 cyclists?*

Solution 21.2: Let's start with five time-trials with no cyclist being in more than one of these five initial time-trials. Let the rankings be $\langle A1, A2, A3, A4, A5\rangle$, $\langle B1, B2, B3, B4, B5\rangle$, $\langle C1, C2, C3, C4, C5\rangle$, $\langle D1, D2, D3, D4, D5\rangle$, and $\langle E1, E2, E3, E4, E5\rangle$, where the first cyclist in each sequence is the fastest. Note that we can eliminate $A4, A5, B4, B5, C4, C5, D4, D5, E4$, and $E5$ at this stage.

Now, we race the winners from each of the initial time-trials. Without loss of generality, assume the outcome is $\langle A1, B1, C1, D1, E1\rangle$. At this point, we can eliminate $D1$ and $E1$ as well as $D2$ and $D3$ and $E2$ and $E3$. Furthermore, since $C1$ was third, $C2$ and $C3$ cannot be in the top three; Similarly, $B3$ cannot be a contender.

We need to find the best and the second best from $A2, A3, B1, B2$, and $C1$, which we can determine with one more time-trial. Therefore seven time-trials are enough.

Note that we need six time-trials to determine the overall winner, and the sequence of time-trials to determine the winner is essentially unique—if some cyclists did not participate in the first five time-trials, he would have to participate in the sixth one. But then one of the winners of the first five time-trials would not participate in the sixth time-trial and he might be the overall winner. The first six time-trials do not determine the second and the third fastest cyclists, hence a seventh race is necessary.

Problem 21.3, *pg. 164*: *Prove that there exists a place such that Albert is at that place at the same time on Sunday as he was on Saturday.*

Solution 21.3: The easiest way to prove this is to imagine another hiker (call him Max) descending the mountain on Saturday, in exactly the same fashion as Albert did on Sunday. When ascending on Saturday, Albert will pass Max at some time and place—this is the time and place which Albert will be at on Sunday.

Problem 21.4, *pg. 164*: *How would you break a 4×4 bar into 16 pieces using as few breaks as possible?*

Solution 21.4: If the assumption is that once you have broken the bar into two pieces, they become separate problems, then it does not matter what order you do it—you will require 15 total breaks in *any* scenario, since each break increases the number of pieces by 1.

If, on the other hand, the assumption is that the whole bar stays together (as it would if you were breaking it in its wrapper, for instance), then you can do a little better. You could simply break it along all axes (say, first the vertical and then the horizontal) for a total of six breaks.

Problem 21.5, *pg. 164*: *If you want to ensure you do not lose, would you rather be F or S?*

Solution 21.5: Number the coins from 1 to 16. Player F can choose all the even-numbered coins by first picking Coin 16 and then always picking the coin at an even index at one of the two ends. For example, if Player S chooses Coin 1, then in the next turn, Player F chooses Coin 2. If Player S chooses Coin 15, then F chooses Coin 14 in the next turn. In this fashion, F can always leave an arrangement where S can only choose from odd-numbered coins.

If the value of the coins at even indices is greater than that of the coins at odd indices, F can win by selecting the even indices and vice versa. If the values are the same, he can simply choose either and in each case, he cannot lose. For the configuration in Figure 4.6 on Page 44, the sum of the even-numbered coins is 125¢ and the sum of the odd-numbered coins is 63¢, so F can win by picking up the even-numbered coins.

The code below implements this idea. It entails a single pass through an array, and its time complexity is $O(n)$, where n is the number of coins (assumed to be even). In contrast, the DP algorithm presented in Solution 15.18 on Page 357 has time complexity $O(n^2)$. The benefit of the DP algorithm is that it yields the optimum solution—140¢ instead of 125¢ for this example.

```
// Return 0 means choosing F, and return 1 means choosing S
template <typename CoinType>
int pick_up_coins(const vector<CoinType> &C) {
  int even_sum = 0, odd_sum = 0;
  for (int i = 0; i < C.size(); ++i) {
    if (i & 1) {  // odd
      odd_sum += C[i];
    } else {  // even
      even_sum += C[i];
    }
  }
  return even_sum >= odd_sum ? 0 : 1;
}
```

Problem 21.6, *pg. 164*: *Assuming the players have infinite computational resources at their disposal, who will win $n \times n$ chomp?*

Solution 21.6: The first player can always win. The key observation in this game is that we want to force the play to be symmetrical around the diagonal, i.e., $(0,0),(1,1),\ldots,(n-1,n-1)$ with our opponent forced to be first to break the symmetry. If that is the case, we can follow each of his moves by a matching move reflected in this diagonal which will eventually force him to select the $(0,0)$ space.

The way to force this kind of play is to be the first person to select $(1,1)$—this causes the play area to be just the column $(0,[0,n-1])$ and the row $([0,n-1],0)$ (i.e., an "L" shape). At that point, the first player can successfully mirror any move that the second player makes, forcing the second player to eventually choose $(0,0)$. This strategy is formalized in pseudocode below.

```
WinChomp():
  You choose square (1,1);
```

```
Until you win:
  Wait for your opponent to choose square (i,j);
  You choose square (j,i);
```

Problem 21.7, *pg. 165*: *Solve Problem 21.6 on Page 164 if the rectangle is n long along the x-axis, and two long along the y-axis.*

Solution 21.7: Suppose the set of remaining squares are of the form of a rectangle and one additional square (which must be on the lower row) and the second player is to move. The remaining set of squares will be in the form of a rectangle (if the second player plays the lower row) or a rectangle with a set of additional squares on the lower row. In either case, the first player can recreate the state to be a rectangle and one additional square, i.e., the first player can force a win. By playing $(n-1, 1)$ as his initial move, the first player can create this situation and therefore force a win.

Problem 21.8, *pg. 165*: *Solve Problem 21.6 on Page 164 if the rectangle is of dimension $n \times m$.*

Solution 21.8: Suppose the second player has a winning strategy. Suppose the first player chose $(n-1, m-1)$ as his initial choice and the second player countered with position (i, j), leaving the set S of squares. Now, it is the first player's turn and from this set, by hypothesis, the second player can force a win. However the first player could have chosen (i, j) as his initial move and the set of remaining squares would be S (since the square $(n-1, m-1)$ is above and to the right of all other squares) with the second player's turn.

This contradicts the hypothesis that the second player has a winning strategy; therefore the first player must have a winning strategy.

Note that this solution does not give an explicit strategy, unlike Solution 21.6 on the facing page and Solution 21.7 on the current page. The argument used above is sometimes called "strategy stealing." Explicit strategies for $n \times m$ chomp are known for only a few specific cases.

Problem 21.9, *pg. 165*: *Building i has r_i residents, and is at distance d_i from the beginning of the street. Devise an algorithm that computes a distance m from the beginning of the street for the mailbox that minimizes the total distance, that residents travel to get to the mailbox, i.e., minimizes $\sum_{i=0}^{n-1} r_i|d_i - m|$.*

Solution 21.9: Suppose the total number of residents $t = \sum_{i=0}^{n-1} r_i$ is odd. Sort the residents according to their distance from the beginning of the street. The optimum location for the mailbox is at the building corresponding to the resident at the median according to the sort order above. Call this resident the median resident.

The argument is as follows. Each location to the left of the building b the median resident lives in is suboptimum, since we can reduce the total distance by moving the mailbox to b (more residents see a reduced distance than see an increased distance). In the same way, each mailbox location to the right of b is suboptimum.

If the number of residents is even, any location between the buildings that the $\lfloor \frac{t}{2} \rfloor$-th and $(\lfloor \frac{t}{2} \rfloor + 1)$-th residents live in is optimum. (If they live in the same building, which is the optimum building.)

If the buildings are given sorted by distance from the beginning of the street, the median resident can be found in linear time by iterating through the offsets in increasing order, adding the corresponding r_is until the count gets to $\frac{t}{2}$. Note that the time complexity is $O(n)$ where n is the number of buildings, not $O(t)$.

If the distances are not sorted, we can sort in $O(n \log n)$, and then find the median. Alternately, we can find the median using extensions of the randomized median finding algorithm for unweighted data. In particular, the randomized median finding algorithm presented in Solution 11.13 on Page 270 can be made to work in linear time, even for the weighted case.

Variant 21.9.1: Compute a location that minimizes the sum of the squares of the distances traveled by the residents.

Variant 21.9.2: Compute a location that minimizes the maximum distance that any resident travels.

Problem 21.10, *pg. 165*: *Given an instance of the gasup problem, how would you efficiently compute an ample city if one exists?*

Solution 21.10: Consider the thought experiment of starting at an arbitrary city with sufficiently large amount of gas so that we can complete the loop. In this experiment, we note the amount of gas in the tank as the vehicle goes through the loop at each city before loading the gas kept in that city for the vehicle. Let C be a city where the amount of gas in the tank before we refuel at that city is minimum. Call this minimum amount of gas m. Now suppose we pick C as the starting point, and we have no gas. Since we never have less gas than we started with at C, we can complete the journey without running out of gas. The computation to determine C can be easily done in linear time with a single pass over all the cities.

```
template <typename T>
int find_start_city(const vector<T> &G, const vector<T> &D) {
  T carry = 0;
  pair<int, T> min(0, 0);
  for (int i = 1; i < G.size(); ++i) {
    carry += G[i - 1] - D[i - 1];
    if (carry < min.second) {
      min = {i, carry};
    }
  }
  return min.first;
}
```

Problem 21.11, *pg. 166*: *Write a function that takes a nonnegative integer x and returns, as a string, the integer closest to x whose decimal representation is a palindrome. For example,*

given 1224, *you should return* 1221.

Solution 21.11: If x is a palindrome, we simply return x. Assume x is not a palindrome. Treat x as a string of digits. For simplicity, assume x is of even length. Let t be the left half of x, and b be the right half of x. Let s be the smallest palindrome greater than x, and l be the largest palindrome smaller than x.

Conceptually, there are three candidates for the palindrome closest to x. One is the integer which agrees with the first half of x. The other two are formed from $t+1$ and $t-1$. In all three cases, the second half of the candidate is implied by the first half. For the given example, we try 1221, 1331, and 1111. The palindrome formed by mirroring $t+1$ may be strictly greater than s; if so, s is guaranteed to be the mirror of t. A parallel result holds for $t-1$.

For odd length x, we use the same approach, but take t to be the digits to the left of the center digit.

```
unsigned diff(const unsigned &a, const unsigned &b) {
  return a > b ? a - b : b - a;
}

unsigned find_closest_palindrome(const unsigned &x) {
  string str(to_string(x));
  // Make str a palindrome by mirroring the left half to the right half
  copy(str.cbegin(), str.cbegin() + (str.size() >> 1), str.rbegin());

  unsigned mirror_left = stoul(str);
  int idx = (str.size() - 1) >> 1;
  if (mirror_left >= x) {
    // Subtract one from the left half
    while (idx >= 0) {
      if (str[idx] == '0') {
        str[idx--] = '9';
      } else {
        --str[idx];
        break;
      }
    }
    if (str[0] == '0') {  // special case, make the whole string as "99...9"
      str = to_string(stoul(str));  // removes the leading 0
      fill(str.begin(), str.end(), '9');
    }
  } else {  // mirror_left < x
    // Add one to the left half
    while (idx >= 0) {
      if (str[idx] == '9') {
        str[idx--] = '0';
      } else {
        ++str[idx];
        break;
      }
    }
  }

  // Make str a palindrome again by mirroring the left half to the right half
```

```
  copy(str.cbegin(), str.cbegin() + (str.size() >> 1), str.rbegin());
  return diff(x, mirror_left) < diff(x, stoul(str)) ?
         mirror_left : stoul(str);
}
```

ϵ-Variant 21.11.1: Find the palindrome p closest to a nonnegative integer x, subject to the constraint that $p \neq x$. For example, if $x = 999$, then $p = 1001$.

Problem 21.12, *pg. 166*: *Given an array A with n elements, compute* $\max_{j=0}^{n-1} \frac{\prod_{i=0}^{n-1} A[i]}{A[j]}$ *in $O(n)$ time without using division. Can you design an algorithm that runs in $O(1)$ space and $O(n)$ time? Array entries may be positive, negative, or 0.*

Solution 21.12: Let $L_p = \prod_{i=0}^{p} A[i]$ and $R_p = \prod_{j=p}^{n-1} A[j]$. Observe computing L_p and R_p individually takes p and $(n-1)-p$ multiplications, respectively; however, we can compute all the L_p and R_p using $2(n-1)$ multiplications, since $L_p = L_{p-1}A[p]$, and $R_p = A[p]R_{p+1}$. The product of all elements except the i-th one is simply $L_{i-1}R_{i+1}$, hence we can compute these n products using n multiplications, once we have L and R computed. Finding the largest product is simply an iteration with compare and swap in the loop. The time complexity is $O(n)$ and the solution uses two arrays of length n each.

```
template <typename T>
T find_biggest_n_1_product(const vector<T> &A) {
  // Build forward product L, and backward product R
  vector<T> L, R(A.size());
  partial_sum(A.cbegin(), A.cend(), back_inserter(L), multiplies<T>());
  partial_sum(A.crbegin(), A.crend(), R.rbegin(), multiplies<T>());

  // Find the biggest product of (n - 1) numbers
  T max_product = numeric_limits<T>::min();
  for (int i = 0; i < A.size(); ++i) {
    T forward = i > 0 ? L[i - 1] : 1;
    T backward = i + 1 < A.size() ? R[i + 1] : 1;
    max_product = max(max_product, forward * backward);
  }
  return max_product;
}
```

It is possible to solve this problem with only $O(1)$ additional storage, with a sophisticated case analysis. Suppose $A[i] \neq 0$ for all i. If A contains an odd number of negative entries, the optimum product is formed when we exclude the biggest negative entry. Otherwise, suppose A contains an even number of negative entries. If some entries are positive, the optimum product is achieved when we exclude the smallest positive entry; otherwise, we exclude the smallest negative number.

Now suppose two or more zeros are present in A. Then the product of any $n-1$ entries is always 0. Suppose there is exactly one zero. If A contains an odd number of negative numbers, the optimum product is zero; otherwise it is the product of

all elements excluding the zero. The code can be readily implemented with a small number of traversals of the array, leading to an $O(n)$ time complexity.

```
template <typename T>
T find_biggest_n_1_product(const vector<T> &A) {
  int zero_count = 0, pos_count = 0, neg_count = 0;
  int zero_idx = -1, s_neg_idx = -1, b_neg_idx = -1, s_pos_idx = -1;

  for (int i = 0; i < A.size(); ++i) {
    if (A[i] < 0) {
      ++neg_count;
      if (s_neg_idx == -1 || A[i] < A[s_neg_idx]) {
        s_neg_idx = i;
      }
      if (b_neg_idx == -1 || A[b_neg_idx] < A[i]) {
        b_neg_idx = i;
      }
    } else if (A[i] == 0) {
      zero_idx = i, ++zero_count;
    } else {  // A[i] > 0
      ++pos_count;
      if (s_pos_idx == -1 || A[i] < A[s_pos_idx]) {
        s_pos_idx = i;
      }
    }
  }

  // Try to find a number whose elimination could maximize the product of
  // the remaining (n - 1) numbers.
  int x;  // stores the idx of eliminated one
  if (zero_count >= 2) {
    return 0;
  } else if (zero_count == 1) {
    if (neg_count & 1) {
      return 0;
    } else {
      x = zero_idx;
    }
  } else {
    if (neg_count & 1) {  // odd number negative
      x = b_neg_idx;
    } else {  // even number negative
      if (pos_count > 0) {
        x = s_pos_idx;
      } else {
        x = s_neg_idx;
      }
    }
  }

  T product = 1;
  for (int i = 0; i < A.size(); ++i) {
    if (i != x) {
      product *= A[i];
    }
```

```
  }
  return product;
}
```

Variant 21.12.1: Let A be as above. Compute an array B where $B[i]$ is the product of all elements in A except $A[i]$. You cannot use division. Your time complexity should be $O(n)$, and you can only use $O(1)$ additional space.

Problem 21.13, *pg. 166*: *Given c cases and d drops, what is the maximum number of floors that you can test in the worst-case?*

Solution 21.13: Let $F(c,d)$ be the maximum number of floors we can test with c identical cases and at most d drops. We know that $F(1,d) = d$. Suppose we know the value of $F(i,j)$ for all $i \leq c$ and $j \leq d$.

If we are given $c+1$ cases and d drops we can start at floor $F(c,d-1)+1$ and drop a case. If the case breaks, then we can use the remaining c cases and $d-1$ drops to determine the floor exactly, since it must be in the range $[1, F(c,d-1)]$. If the case did not break, we proceed to floor $F(c,d-1)+1+F(c+1,d-1)$.

Therefore F satisfies the recurrence

$$F(c+1,d) = F(c,d-1)+1+F(c+1,d-1).$$

We can compute F using DP as below:

```
int get_height_helper(vector<vector<int>> &F, const int &c, const int &d) {
  if (d == 0) {
    return 0;
  } else if (c == 1) {
    return d;
  } else {
    if (F[c][d] == -1) {
      F[c][d] = get_height_helper(F, c, d - 1) +
                get_height_helper(F, c - 1, d - 1) + 1;
    }
    return F[c][d];
  }
}

int getHeight(const int &c, const int &d) {
  vector<vector<int>> F(c + 1, vector<int>(d + 1, -1));
  return get_height_helper(F, c, d);
}
```

Variant 21.13.1: How would you compute the minimum number of drops needed to find the breaking point from 1 to F floors using c cases?

Variant 21.13.2: Men numbered from 1 to n are arranged in a circle in clockwise order. Every k-th man is removed, until only one man remains. What is the number

of the last man?

Problem 21.14, *pg. 166*: *Moles are numbered from 0 to $n-1$. Mole m has a set of neighboring moles. Whacking m when it is up results in it and all of its neighbors flipping state. Given a set of moles, the neighbors for each mole, and an initial assignment of up/down states for each mole, compute a sequence of whacks (if one exists) that results in each mole being in the down state.*

Solution 21.14: First we make the observation that any pair of whacks commute, that is that the state resulting from a whack to mole m_i followed by a whack to mole m_j is the same as the state resulting from whacking m_j followed by m_i. This generalizes to arbitrary sequences of whacks, which in turn implies that a state can be achieved from an initial state s_0 iff it can be achieved by whacking each mole at most once.

Introduce a Boolean variable x_i for Mole i—this variable indicates whether the mole is to be whacked. Observe that the state of Mole i after the whacks encoded by the x_is is $y_i = x_i \oplus x_{n_0^i} \oplus x_{n_1^i} \cdots \oplus x_{n_{i_{k-1}}^i}$. Therefore the problem reduces to computing an assignment to the x_is that sets each y_i to 0.

The standard approach to solving linear equations of the form $y = Ax$ is Gaussian elimination. Iteratively remove variable x_i from all equations after the i-th one. This results in an equation in one unknown, which is then solved, and iteratively substituted back into the previous equations.

```
void Eliminate_rows(vector<vector<bool>> &B, const int &i, const int &j) {
  // Use B[i] to eliminate other rows' entry j
  for (int a = 0; a < B.size(); ++a) {
    if (i != a && B[a][j]) {
      for (int b = 0; b < B[i].size(); ++b) {
        B[a][b] = B[a][b] ^ B[i][b];
      }
    }
  }
}

vector<bool> Gaussian_elimination(const vector<vector<bool>> &A,
                                  const vector<bool> &y) {
  vector<vector<bool>> B(A);
  for (int i = 0; i < B.size(); ++i) {
    B[i].push_back(y[i]);
  }

  for (int i = 0; i < B.size(); ++i) {
    // Find the coefficient starting with 1
    int idx = i;
    for (int j = i + 1; j < B.size(); ++j) {
      if (B[j][i]) {
        idx = j;
        break;
      }
    }
    swap(B[i], B[idx]);

```

```
    // Perform elimination except i-th row
    if (B[i][i]) {
      Eliminate_rows(B, i, i);
    }
  }

  for (int i = B.size() - 1; i >= 0; --i) {
    if (B[i][i] == false) {
      bool have_coefficient = false;
      for (int j = i + 1; j < A.size(); ++j) {
        if (B[i][j]) {
          Eliminate_rows(B, i, j);
          have_coefficient = true;
          swap(B[i], B[j]);  // row permutation
          break;
        }
      }

      if (have_coefficient == false && B[i].back() == true) {
        cout << "No solution." << endl;
        return {};
      }
    }
  }

  vector<bool> x;
  for (int i = 0; i < B.size(); ++i) {
    x.push_back(B[i].back());
  }
  return x;
}
```

Problem 21.15, *pg. 167*: *Let F be an $n \times n$ Boolean 2D array representing the "knows" relation for n people; $F[a][b]$ is* `true` *iff a knows b, and $F[a][a]$ is always* `false`*. Design an $O(n)$ algorithm to find the celebrity.*

Solution 21.15: We start by checking $F[0][1]$, i.e., the relation between Person 0 and Person 1. Assuming that we are checking $F[i][j]$ where $i < j$, the key idea here is that if $F[i][j]$ is `false`, we know that j is not the celebrity and i is still a possible celebrity candidate, allowing us to eliminate j from the set of celebrity candidates by advancing from $F[i][j]$ to $F[i][j+1]$. If $F[i][j]$ is `true`, we know that i is not the celebrity, and for all $j' < j$, j' is not a celebrity because $F[i][j']$ must be `false`, allowing us to advance from $F[i][j]$ to $F[j][j+1]$ since $i < j$. We eliminate one candidate each step in $O(1)$ time which gives us a $O(n)$ time algorithm.

```
int celebrity_finding(const vector<vector<bool>> &F) {
  // Start checking the relation from F[0][1]
  int i = 0, j = 1;
  while (j < F.size()) {
    if (F[i][j] == true) {
      i = j++;  // all candidates j' < j are not celebrity candidates
    } else {  // F[i][j] == false
```

```
        ++j;  // i is still a celebrity candidate but j is not
      }
    }
    return i;
}
```

Variant 21.15.1: Solve the same problem when the knows relation is not reflexive, nor anti-reflexive, i.e., some people may know themselves, and others may not.

Problem 21.16, *pg. 167*: *Six guests attend a party. Any two guests either know each other or do not know each other. Prove that there exists a subset of three guests who either all know each other or all do not know each other.*

Solution 21.16: This problem can be modeled using undirected graphs where vertices correspond to guests. Add an edge between each pair of guests. Color an edge between a pair of guest "blue" if they are friends, otherwise, color it "red".

Then the theorem is equivalent to the claim that in any clique on six vertices, where each edge is either blue or red, there exists a subset of three vertices, all connected by edges of the same color.

Choose any vertex v. Examine the five edges having v as an endpoint. By the pigeon-hole principle, there must be at least three edges which are of the same color c. Let $(v, \alpha), (v, \beta)$, and (v, γ) be three such edges. Now, either there is an edge colored c between α, β, and γ, in which case v and the two vertices in α, β, and γ that are connected by a c-colored edge are three vertices all connected by c-colored edges, or there is no such edge. In the latter case, α, β, and γ are themselves connected by edges that are of the same color.

Ramsey's theorem is illustrated in Figure 21.17. In each graph, the three shaded nodes are either all connected or all disconnected.

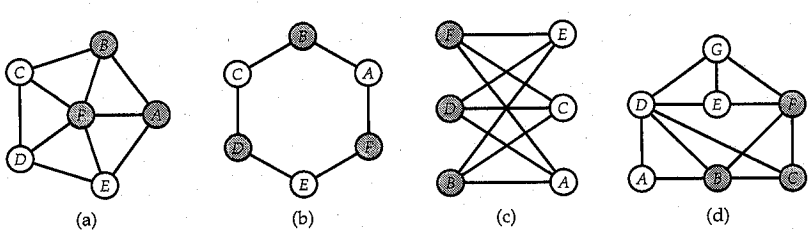


Figure 21.17: Representative graphs on six vertices.

Problem 21.17, *pg. 167*: *Prove that every tournament has a Hamiltonian path, i.e., a path that includes each vertex exactly once.*

Solution 21.17: We use induction on the number of vertices. Specifically, the induction hypothesis is that every tournament on n vertices has a Hamiltonian path.

Proof:

The base case, namely $n = 1$, is trivial. For the induction step, let T be a tournament on $n + 1$ vertices, and v_n be any vertex in T. Note that removing a vertex together with its incoming and outgoing edges from any tournament on $n + 1$ vertices leaves a tournament on n vertices.

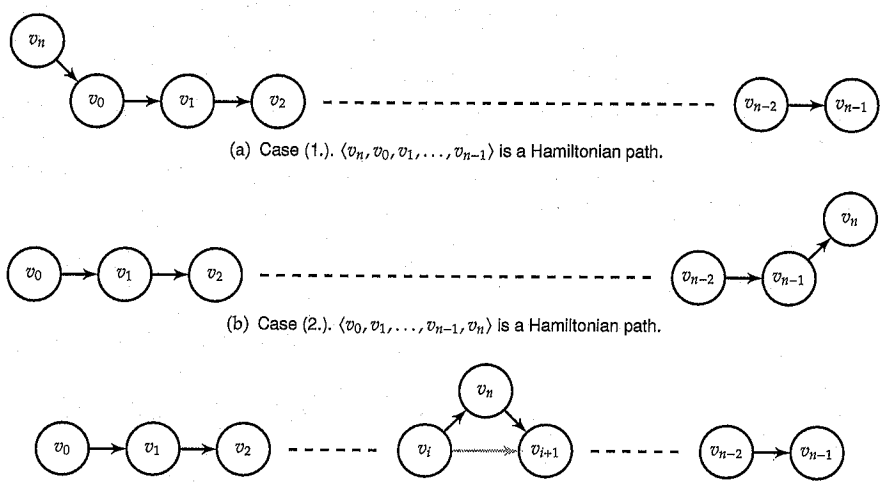
Let $\langle v_0, v_1, \ldots, v_{n-1} \rangle$ be a Hamiltonian path in the graph resulting from the removal of v_n from T. Its existence is guaranteed by the induction hypothesis. There exist three possibilities:

(1.) An edge exists from v_n to v_0. Then $\langle v_n, v_0, v_1, \ldots, v_{n-1} \rangle$ is a Hamiltonian path in T, so induction goes through.

(2.) An edge exists from v_{n-1} to v_n. Then $\langle v_0, v_1, \ldots, v_{n-1}, v_n \rangle$ is a Hamiltonian path in T, so induction goes through.

(3.) No edge exists from v_n to v_0 and no edge exists from v_{n-1} to v_n. Then there must be an edge from v_0 to v_n, and an edge from v_n to v_{n-1}. Let i be the maximum over all j such that (v_j, v_n) is an edge in T. Since we have ruled out $i = n - 1$ above, there must be an edge from v_n to v_{i+1}. The path $\langle v_0, v_1, \ldots, v_i, v_n, v_{i+1}, \ldots v_{n-1} \rangle$ is a Hamiltonian path in T, so induction goes through.

The different cases are illustrated in Figure 21.18. The proof can be shortened to



(a) Case (1.). $\langle v_n, v_0, v_1, \ldots, v_{n-1} \rangle$ is a Hamiltonian path.

(b) Case (2.). $\langle v_0, v_1, \ldots, v_{n-1}, v_n \rangle$ is a Hamiltonian path.

(c) Case (3.). $\langle v_0, v_1, \ldots, v_i, v_n, v_{i+1}, \ldots v_{n-1} \rangle$ is a Hamiltonian path.

Figure 21.18: Case analysis view graphically.

the following: choose $i \in \{0, 1, 2, \ldots, n-1\}$ to be maximal such that for every $j \leq i$ there is an edge from v_j to v_n; Then the sequence $\langle v_0, v_1, \ldots, v_i, v_n, v_{i+1}, \ldots, v_{n-1} \rangle$ is a Hamiltonian cycle.

Problem 21.18, *pg. 167*: *Design an algorithm which takes the preference lists of the students and the professors and pairs students one-to-one with professors subject to the constraint that*

there do not exist student-professor pairings $(s0, p0)$ *and* $(s1, p1)$ *such that* $s0$ *prefers* $p1$ *to* $p0$ *and* $p1$ *prefers* $s0$ *to* $s1$. *(The preferences of* $p0$ *and* $s1$ *are not important.)*

Solution 21.18: This problem can be solved using a "proposal algorithm". We will refer to the professor a student is paired with as his adviser. Each student who does not have an adviser "proposes" to the professor he likes best to whom he has not yet proposed.

Each professor then considers all the students who have sent him proposals and tells the student in this set that he most prefers "lets talk more", and he says "no" to all the other student who proposed him.

In each subsequent round, each student who does not have an adviser proposes to one professor to whom he has not yet proposed. This is done regardless of whether the professor has already been matched. The professor once again replies with a single "lets talk more", rejecting the rest. Note that a professor who has already said "lets talk more" to a student may switch to another student he prefers more, leaving the first student unmatched.

This algorithm has two key properties, which immediately yield its correctness.

- It converges to a state where everyone is paired: Once a professor begins talking to a student, he always has a student to talk to. There cannot be a professor and a student both unpaired since the student must made a request to that professor at some point—a student will eventually propose to everyone, if necessary and being unpaired, the professor would have accepted.
- The pairings are stable: Let Riemann be a student and Gauss be a professor. Suppose Gauss is talking to Dedekind and Riemann is talking to Eisenstein. Upon completion of the algorithm, it is not possible for both Riemann and Gauss to prefer each other over their current pairings. If Riemann prefers Gauss to Eisenstein, he must have asked Gauss before he asked Eisenstein. If Gauss accepted Riemann, but is not paired to Riemann at the end, he must have rejected Riemann for a student he preferred more than Riemann and therefore does not like Riemann more than Dedekind.

```
vector<pair<int, int>> find_stable_assignment(
    const vector<vector<int>> &professor_preference,
    const vector<vector<int>> &student_preference) {
  queue<int> free_student;  // stores currently free students
  for (int i = 0; i < student_preference.size(); ++i) {
    free_student.emplace(i);
  }

  // Records the professor each student have asked
  vector<int> student_pref_idx(student_preference.size(), 0);
  // Records each professor currently student choice
  vector<int> professor_choice(professor_preference.size(), -1);

  while (free_student.empty() == false) {
    int i = free_student.front();   // free student
    int j = student_preference[i][student_pref_idx[i]];   // target professor
    if (professor_choice[j] == -1) {  // this professor is free
      professor_choice[j] = i;
```

```
      free_student.pop();
    } else {  // this professor has student now
      int original_pref = find(professor_preference[j].cbegin(),
                               professor_preference[j].cend(),
                               professor_choice[j]) -
                          professor_preference[j].cbegin();
      int new_pref = find(professor_preference[j].cbegin(),
                          professor_preference[j].cend(), i) -
                     professor_preference[j].cbegin();
      if (new_pref < original_pref) {   // this professor prefers the new one
        free_student.emplace(professor_choice[j]);
        professor_choice[j] = i;
        free_student.pop();
      }
    }
    ++student_pref_idx[i];
  }

  vector<pair<int, int>> match_result;
  for (int j = 0; j < professor_choice.size(); ++j) {
    match_result.emplace_back(professor_choice[j], j);
  }
  return match_result;
}
```

Problem 21.19, *pg. 168*: *Design an algorithm for pairing bidders with celebrities to maximize the revenue from the dance. Each celebrity cannot dance more than once, and each bidder cannot dance more than once. Assume that the set of celebrities is disjoint from the set of bidders. How would you modify your approach if all bids were for the same amount? What if celebrities and bidders are not disjoint?*

Solution 21.19: The problem can directly be mapped into the weighted bipartite matching problem. Bidders and celebrities constitute the left and right vertices; an edge exists from b to c iff b has offered money to dance with c, and the weight of an edge is the amount offered for the dance. It can be solved using specialized algorithms, flow network, or linear programming.

If the bids are all in the same amount, the problem is that of unweighted bipartite matching. If the requirement that bidders and celebrities be distinct is dropped, the problem becomes a weighted matching problem in a general graph. Both of these variants are solvable in polynomial time.

Problem 21.20, *pg. 168*: *Suppose two squares of opposite colors are removed from a chessboard. Design an algorithm for finding a way to cover the remaining squares using* 31 *dominoes, if a covering exists.*

Solution 21.20: Model the original chessboard as a bipartite graph B, with 32 white vertices on the left and 32 black vertices on the right. Vertices correspond to squares. Put an edge between vertices whose corresponding squares are adjacent. In particular, corner vertices are connected to two edges; edge vertices are connected to three

edges; and remaining vertices are connected to four edges.

Removing a white and a black square from the chessboard corresponds to removing a left and a right vertex, together with the edges they are connected to. Call the new graph B'.

Observe that a single domino placed on the chessboard covers adjacent white and black vertices. Therefore a valid placement of dominoes corresponds to a set of edges, no two of which share a vertex. Such sets are commonly referred to as matchings.

The desired covering corresponds to a matching in B' containing 31 edges, which is the maximum possible. Hence we can solve our problem using well-known algorithms finding a maximum matching in a bipartite graph.

Alternately, it is fairly simple to construct a Hamiltonian cycle in B—a sequence $\langle v_0, v_1, \ldots, v_{63}, v_0 \rangle$, such that for all $i, 0 \leq i \leq 63$, $(v_i, v_{i+1 \bmod 64})$ is an edge. If we remove a single black and a single white vertex, we can construct a matching that covers all the remaining vertices by beginning immediately after the removed vertices and continuing along the edges in the Hamiltonian cycle. Correctness follows from the fact that there will be an even number of vertices on the subpaths in the Hamiltonian cycle that result on the deletion of the two vertices. This construction is illustrated in Figure 21.19. Therefore, regardless of the white and vertices that are removed, there will always be a suitable covering of the chessboard. (This is not apparent from the maximum matching formulation.)

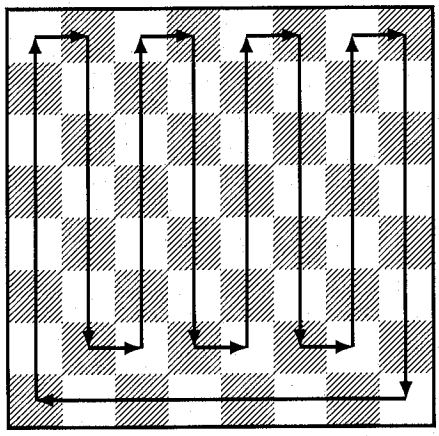


Figure 21.19: Hamiltonian cycle in a chessboard.

Variant 21.20.1: Is it true that when *any* single square is removed from a $2^k \times 2^k$ chessboard, the remaining $2^k \times 2^k - 1$ squares can be tiled with triominoes? (See also the discussion of mutilated chessboards on Page 30.)

Problem 21.21, *pg. 168*: *This problem is a continuation of Problems 13.6 on Page 100*

and 16.7 on Page 135. Design an efficient algorithm for computing the minimum number of subsets of teams so that (1.) the teams in each subset can be organized to appear in a single photograph without violating the placement constraint, and (2.) each team appears in exactly one subset.

Solution 21.21: In Solution 16.7 on Page 382, we showed how to model the problem using a DAG, with each vertex corresponding to a team. The current problem is asking for a minimum cardinality set of paths in this DAG such that each vertex appears on some path.

This is a classic problem that can be solved using maximum bipartite matching as a subroutine. Let $G = (V, E)$ be a DAG. Construct a bipartite graph B from G as follows. Each vertex $v \in V$ is represented with two vertices v and v'. For each (directed) edge $(u, v) \in E$ add an (undirected) edge (u, v') to B.

We now prove a one-to-one correspondence between matchings in B and vertex-disjoint paths that cover all vertices in G.

Proof:

Let Π be a set of vertex-disjoint paths. Initialize M to the empty set. For each $\langle u, v, w, \ldots, x, y\rangle \in \Pi$ add the edges $(u, v'), (v, w'), \ldots, (x, y')$ to M. We claim M is a matching. For two edges of the form (a, b') and (a, c') to be present in M, the edges (a, b) and (a, c) must have been present in two different paths. (The paths must be different since G is a DAG.) This contradicts the assumption that the paths in Π are vertex-disjoint. A similar argument holds for edges of the form (a', b) and (a', c).

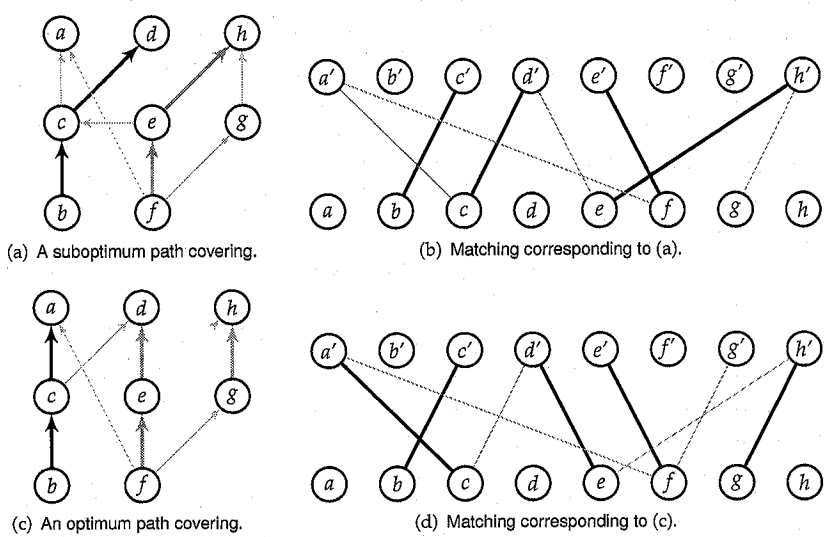
Conversely, let M be a matching in B. Construct a set of vertex-disjoint paths Π that cover G from M as follows. Initialize S to the empty set. While $S \neq V$, choose a vertex $v \notin S$ and iteratively create a path p in G including v as follows. If v is matched in M with say a', add (v, a) to p and continue the construction with a, stopping when an unmatched vertex is encountered. Additionally, if v' is matched with b add (b, v) to p and continue the construction with b. Since M is a matching, we will never add the same vertex to two paths. Furthermore, since G is a DAG, the construction of p will always end.

Observe that $|\Pi| = |V| - |M|$, since each edge in a path corresponds to a matched edge in B. Therefore we can find a minimum vertex-disjoint set of paths that covers G by finding a maximum matching in B.

The proof shows that each unmatched vertex v corresponds to the last vertex in a path, and each unmatched vertex v' corresponds to the first vertex in a path. Therefore the number of unmatched vertices in B is twice the number of paths in the corresponding set of vertex-disjoint paths in G. Therefore, the minimum number of vertex-disjoint paths covering G can be determined by computing a maximum matching in B.

The construction is illustrated in Figure 21.20 on the facing page. Figure 21.20(a) on the next page is a DAG covered by four paths, $\langle b, c, d\rangle$, $\langle f, e, h\rangle$, $\langle a\rangle$, and $\langle g\rangle$. We use thick edges to denote edges in paths; thin edge are not part of the path covering. Different paths are shaded differently. The corresponding bipartite graph and

matching are shown in Figure 21.20(b); dark edges are the ones in the matching. The unmatched vertices on the lower row are the vertices that end paths, and the unmatched on the upper row are the vertices that path begin from. An augmenting path for a matching is a path beginning and ending at unmatched vertices which alternates between unmatched and matched edges. The existence of an augmenting path $\langle a', c, d', e, h', g \rangle$, indicates that the matching is suboptimum. The path $\langle a', c, d', e, h', g \rangle$ is an augmenting path, demonstrating that the matching in Figure 21.20(b) is suboptimum. Figure 21.20(c) illustrates the same DAG covered with three paths, $\langle b, c, a \rangle$, $\langle f, e, d \rangle$, and $\langle g, h \rangle$. The corresponding bipartite graph and matching are shown in Figure 21.20(d). The matching in Figure 21.20(d) is maximum, since there is no augmenting path for it.



(a) A suboptimum path covering.

(b) Matching corresponding to (a).

(c) An optimum path covering.

(d) Matching corresponding to (c).

Figure 21.20: Relationship between paths and matchings.

Variant 21.21.1: Define a subset S of $\mathcal{Z}^+$ to be division-free if $\nexists x, y \in S$ such that $x \bmod y = 0$. How would you compute a maximum cardinality subset division-free subset of $\mathcal{Z}_n$?

Variant 21.21.2: Define G to be the graph on $V = \{0, 1, \ldots, m-1\} \times \{0, 1, \ldots, n-1\}$, and $E = \{((i, j), (i+1, j)) \mid 0 \leq i < m-1, 0 \leq j \leq n-1\} \cup \{((i, j), (i, j+1)) \mid 0 \leq i \leq m-1, 0 \leq j < n-1\}$. Let S be a subset of the vertices. How would you compute the minimum number of paths that begin at $(0, 0)$ and end at $(m-1, n-1)$ and include each vertex in S at least once?

Problem 21.22, *pg. 168*: *Let A and B be rooted trees. Design a polynomial time algorithm for computing a largest common rooted subtree of A and B.*

Solution 21.22: We use DP to build a map L from pairs $u \in A$ and $v \in B$ to the size of a largest subtree rooted specifically at u and at v.

Denote by $C(x)$ the set of children nodes of x. Suppose we know $L(u', v')$ for all $u' \in C(u)$ and $v' \in C(v)$. Finding the largest common subtree rooted at u and v entails finding a one-to-one mapping m from $M_u \subset C(u)$ to $M_v \subset C(v)$ that maximizes $\sum_{u' \in M_u} L(u', m(u'))$.

Form a complete bipartite graph B on $C(u) \cup C(v)$, with the weight of edge (u', v') being $L(u', v')$. The desired mapping is simply a maximum weighted bipartite matching in B.

The complexity of this procedure is polynomial, but with a high degree. Given a bipartite graph with $|V|$ vertices and $|E|$ edges, a maximum weighted matching can be computed in $O(|V|^2|E|)$ time using max-flow algorithms. This can be improved using specialized matching algorithms to $O(|V||E| + |V|^2 \log |V|)$. In our setting $|E| = \Theta(|V|^2)$, since $L(x, y) \geq 1$ for all x, y. If A and B each have n nodes, the weighted bipartite matching algorithm may be called with $|V| = \Theta(n)$, and $O(n^2)$ such calls are made, leading to a $O(n^5)$ time bound, when the best weighted bipartite matching algorithm is used. In practice it is unlikely that the matching routine will be called n^2 times with large bipartite graphs.

Variant 21.22.1: Given two binary trees, compute the largest subtree that is contained in both. Ignore node contents.

Variant 21.22.2: Given two binary trees with values stored at nodes, compute the largest subtree that is contained in both. Isomorphic nodes must have the same stored values.

Problem 21.23, *pg. 169*: *Consider a league in which teams are numbered from 0 to $n-1$. At a certain point in the season, Team i has won W_i games, and has $R_{i,j}$ games remaining with Team j. Each game will end in a win for one team and a loss for the other team. Show how the problem of determining whether Team a is mathematically eliminated can be solved using maximum flow.*

Solution 21.23: The most wins Team a can end the season with is $\mu_a = W_a + \sum_{j \neq a} R_{a,j}$. The idea is to use max-flow to check if it is not possible for all other teams to simultaneously win no more than μ_a games, in which case a is eliminated.

We define an instance of maximum flow as follows. Let $\{s\} \cup \{r_{i,j} \mid i \neq a, j \neq a, \text{ and } i < j\} \cup \{v_i \mid i \neq a\} \cup \{t\}$ be a set of vertices. Add an edge from s to each vertex of the form $r_{i,j}$ with capacity $R_{i,j}$. Add edges from $r_{i,j}$ to v_i and to v_j with capacity ∞ for each vertex of the form $r_{i,j}$. Add an edge from each vertex v_i to t with capacity $\mu_a - W_i$. This construction is illustrated in Figure 21.21 on the next page.

An integral flow f from s can be interpreted as follows. The flow through $(s, (r_{i,j}))$ is split into flow through v_i and v_j; it is the assignment of $f(s, (r_{i,j}))$ victories across Team i and Team j. The maximum number of victories that Team i can have without eliminating a is $\mu_a - W_i$; this is captured by the capacities on edges of the form (v_i, t).

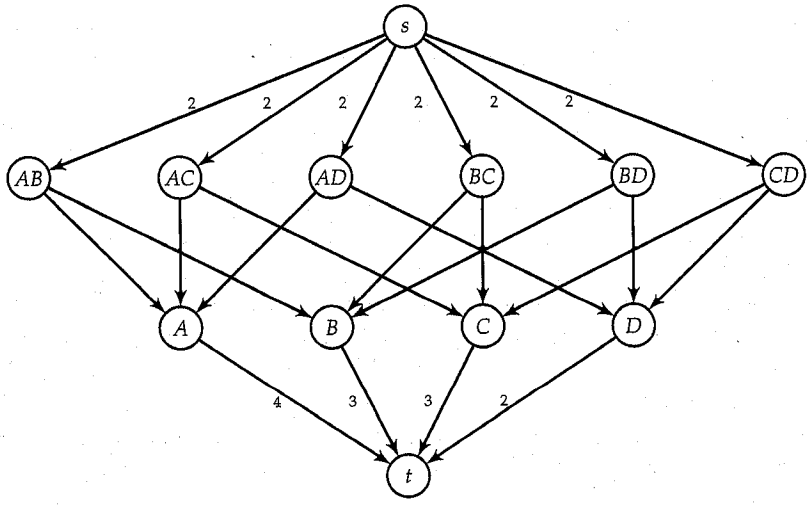


Figure 21.21: Maximum flow instance for the example described on the current page. Nodes immediately below s correspond to matches between pairs of teams $\{A, B, C, D\}$. Edges that are not labeled with a capacity have infinite capacity.

Suppose there exists a flow that saturates all the edges emanating from s, i.e., has value $\sum_{\{(i,j)|i<j \text{ and } i,j\neq a\}} R_{i,j}$. This must be a maximum flow, since it is the capacity of all the edges emanating from s. By the reasoning above, this flow assigns outcomes to all the games that remain to be played that do not involve a, and by the (v_i, t) edge capacity constraint, this assignment does not lead to the elimination of a.

Conversely, suppose a is not eliminated because of some combination of outcomes for the games. These outcomes yield a flow that has value $\sum_{\{(i,j)|i<j \text{ and } i,j\neq a\}} R_{i,j}$ which must satisfy the capacity constraints (since there are $R_{i,j}$ games remaining between Team i and Team j, and Team i wins no more than $\mu_a - W_i$ games).

Putting the two arguments together, we have proved that the network has a maximum flow of value $\sum_{\{(i,j)|i<j \text{ and } i,j\neq a\}} R_{i,j}$ iff a is not eliminated.

As a concrete example, consider five teams, A, B, C, D, and E. We want to check if Team E is eliminated. Two games remain to be played between each pair in $\{A, B, C, D\}$, and E has a total of five games to play against the remaining four teams. Currently, E has one victory less than A, two fewer victories than B and C, and three fewer victories than D.

The technique we presented above yields the maximum flow problem in Figure 21.21, and E is not eliminated iff there exists a flow that saturates all the edges coming out of s. The following flow function achieves this: $f(AB, A) = 1, f(AB, B) = 1, f(AC, A) = 1, f(AC, C) = 1, f(AD, A) = 2, f(AD, D) = 0, f(BC, B) = 0, f(BC, C) = 2, f(BD, B) = 2, f(BD, D) = 0, f(CD, C) = 0, f(CD, D) = 2$, with the remaining values for f implied by those above.

Variant 21.23.1: Let G be a DAG in which each vertex v has an associated value

$c(v) \in \mathcal{Z}$. How would you select a subset S of vertices having maximum value, subject to the constraint that if $v \in S$ and $(u, v) \in E$ then $u \in S$?

Problem 21.24, *pg. 169*: *Design an efficient algorithm for computing a rounding of a matrix, if one exists.*

Solution 21.24: Clearly, a necessary condition for a rounding of A to exist is that each row and each column of A sums to an integer. We will show that this condition is also sufficient via a constructive argument.

We formulate the problem as a variant of the max-flow problem, specifically, as an instance of the circulation with demands problem.

We define a directed graph $G = (V, E)$ as follows. The vertex set $V = \{s, t, u_0, u_1, \ldots, u_{m-1}, v_0, v_1, \ldots, v_{n-1}\}$. Add directed edges (s, u_i), (v_j, t), (u_i, v_j), and (t, s) for all $0 \le i \le m-1$, $0 \le j \le n-1$. Assign a lower bound ("demand") $L(x, y)$ and upper bound ("capacity") $U(x, y)$ to edges as follows: $L(s, u_i) = U(s, u_i) = \sum_{j=0}^{n-1} A[i, j]$, $L(v_j, t) = U(v_j, t) = \sum_{i=0}^{m-1} A[i, j]$, $L(u_i, v_j) = \lfloor A[i, j] \rfloor$, $U(u_i, v_j) = \lceil A[i, j] \rceil$, $L(t, u) = 0$, and $U(t, u) = \infty$.

We know that we can assign flows through the edges of G that satisfy the lower and upper bound constraints by simply setting the flow through each (u_i, v_j) to $A[i, j]$, with the flow through the remaining edges implied by this assignment. Hence the circulation problem is feasible. The circulation integrality theorem says that if an instance of the circulation problem has integral lower and upper bounds for each edge, then the problem is feasible iff there exists a circulation with integer flows. The proof is constructive, hence applying the circulation algorithm yields the desired rounding.

Problem 21.25, *pg. 170*: *What follows after the explorer visits the Isle of Logic? The explorer seems to have added no new knowledge since each inhabitant already knows that some inhabitants have blue eyes and some have green eyes. Why does his observation change the equilibrium?*

Solution 21.25: Consider the case where exactly one person has green eyes. The statement from the explorer would make it clear to the person with green eyes that he has green eyes since nobody else that he sees has green eyes.

Now, suppose two inhabitants have green eyes. The first day, each of these two inhabitants would see exactly one other person with green eyes. Each would see the other person on the second day too, from which they could infer that there must be two inhabitants with green eyes, the second one being themselves. Hence both of them would leave the second day.

Using induction, we can demonstrate that if there are k inhabitants with green eyes, all the green-eyed inhabitants would leave after the k-th assembly.

Proof:

We already saw the base case, $k = 1$. Suppose the induction hypothesis holds for $k - 1$. If the number of inhabitants with green eyes is k, each inhabitant with green eyes would see $k-1$ other inhabitants with green eyes. If at the k-th assembly,

they see that nobody has departed, it would indicate that they themselves have green eyes and hence all the green-eyed inhabitants would leave on the k-th day.

As for the second part of the question, for $k = 1$, it is fairly obvious that the explorer gave new knowledge to the person with green eyes. For other cases, the new information is more subtle. For $k = 2$, the green-eyed inhabitants would be able to infer the color of their eyes on the second day based on the information that everyone on the island knows that there are green-eyed inhabitants and yet no one left. For $k = 3$, they are able to infer because everyone knows that everyone knows that there are green-eyed inhabitants and yet on the second day no one left.

Suppose x is some fact and $E(x)$ represents the fact that everyone knows x to be true. In this case, let g represent the fact that there are some green-eyed inhabitants on the island. Then on the k-th day, all the green-eyed inhabitants would use $E^k(g)$, the k-iteration of the common knowledge, to infer that they have green eyes. Essentially, what the explorer did by announcing the fact in the assembly is that it became "common knowledge", i.e., $E^\infty(g)$ became true.

Problem 21.26, *pg. 170*: *Given a payoff matrix, compute values $p_0, p_1, \ldots, p_{m-1}$ for Player 1 that minimize the maximum payoff for Player 2. Assume Player 2 knows $p_0, p_1, \ldots, p_{m-1}$.*

Solution 21.26: Let $p = [p_0, p_1, \ldots, p_{m-1}]^T$ and $q = [q_0, q_1, \ldots, q_{n-1}]^T$. Denote $A[i][j]$ by $a_{i,j}$. Formally, we want to find a p that minimizes $\max_q p^T Aq$, subject to the constraints that probabilities are nonnegative, and sum up to 1. When Player 2 knows $p_0, p_1, \ldots, p_{m-1}$, there always exists a q maximizing $p^T Aq$ that is a vector of all 0s except for a single 1. The 1 is at an index corresponding to the maximum entry in $p^T A$. (There may be other qs which also maximize, but there is no loss of optimality in focusing on qs of the specified form.) Let e_i denote the n-dimensional vector which is 1 an index i, and 0 everywhere else.

The problem now is to find p that minimizes $\max_{0 \le i \le n-1} p^T Ae_i$. Modeling this optimization problem as a linear program requires a trick to deal with the max operation. The basic idea is that the minimization problem $\min(\max_{0 \le i \le n-1}\{x_0, x_1, \ldots, x_{n-1}\})$ subject to the constraint $C(x_0, x_1, \ldots, x_{n-1})$ can be cast as a minimization problem which does not use the max operator. Specifically, this minimization problem is equivalent to minimizing t subject to $C(x_0, x_1, \ldots, x_{n-1})$ and $t \ge x_i$, for $0 \le i \le n-1$. Therefore the p we seek is the minimizer of t subject to the following constraints:

$$\begin{aligned} t &\ge p^T Ae_i = \sum_{j=0}^{m-1} p_j a_{j,i}, \quad \text{for all } i,\ 0 \le i \le n-1 \\ \sum_{i=0}^{m-1} p_i &= 1 \\ p_i &\ge 0, \quad \text{for all } i,\ 0 \le i \le m-1 \end{aligned}$$

Now this optimization problem is an instance of linear programming, and can be solved using standard algorithms, e.g., simplex.

Variant 21.26.1: How would you efficiently solve for the optimum p subject to the additional constraint that the sum of the k largest values of p is no more than μ? (Adding $\binom{n}{k}$ additional constraints is not efficient.)

Variant 21.26.2: Given a subset of $\mathcal{R}^n$ defined by a set of linear inequalities $Ax \leq b$, find the radius of a largest sphere contained in the subset.

Variant 21.26.3: Suppose $C \in \mathcal{R}^{n \times n}$ and $\rho \in \mathcal{R}$. Let $\mathcal{F} = \{F \in \mathcal{R}^{n \times n} \mid F[i][j] \in [C[i][j] - \rho, C[i][j] + \rho]\}$. The constraints $Ax \leq b$ for all $A \in \mathcal{F}$ define a subset of $\mathcal{R}^n$. Find the radius of a largest sphere contained in this subset.

Part IV

Notation and Index

Notation

To speak about notation as the only way that you can guarantee structure of course is already very suspect.

— E. S. Parker

We use the following convention for symbols, unless the surrounding text specifies otherwise:

i, j, k	nonnegative array indices
f, g, h	function
A	k-dimensional array
L	linked list or doubly linked list
S	set
T	tree
G	graph
V	set of vertices of a graph
E	set of edges of a graph
$\mathcal{E}$	an event from a probability space
u, v	vertex-valued variables
e	edge-valued variable
m, n	number of elements in a collection
x, y	real-valued variables
σ	a permutation

Symbolism	Meaning
$\log_b x$	logarithm of x to the base b
$\lg x$	logarithm of x to the base 2
$\lvert S \rvert$	cardinality of set S
$S \setminus T$	set difference, i.e., $S \cap T'$, sometimes written as $S - T$
$\lvert x \rvert$	absolute value of x
$\lfloor x \rfloor$	greatest integer less than or equal to x
$\lceil x \rceil$	smallest integer greater than or equal to x
$\langle a_0, a_1, \ldots, a_{n-1} \rangle$	sequence of n elements
$a^k, a = \langle a_0, \ldots, a_{n-1} \rangle$	the sequence $\langle a_k, a_{k+1}, \ldots, a_{n-1} \rangle$
$\sum_{R(k)} f(k)$	sum of all $f(k)$ such that relation $R(k)$ is true
$\prod_{R(k)} f(k)$	product of all $f(k)$ such that relation $R(k)$ is true
$\min_{R(k)} f(k)$	minimum of all $f(k)$ such that relation $R(k)$ is true
$\max_{R(k)} f(k)$	maximum of all $f(k)$ such that relation $R(k)$ is true

$\sum_{k=a}^{b} f(k)$	shorthand for $\sum_{a \le k \le b} f(k)$
$\prod_{k=a}^{b} f(k)$	shorthand for $\prod_{a \le k \le b} f(k)$
$\{a \mid R(a)\}$	set of all a such that the relation $R(a)$ = true
$[l, r]$	closed interval: $\{x \mid l \le x \le r\}$
$[l, r)$	half-closed, half-open interval: $\{x \mid l \le x < r\}$
$(l, r]$	half-open, half-closed interval: $\{x \mid l < x \le r\}$
(l, r)	open interval: $\{x \mid l < x < r\}$
$\{a, b, \ldots\}$	well-defined collection of elements, i.e., a set
A_i or $A[i]$	the i-th element of one-dimensional array A
$A[i : j]$	subarray of one-dimensional array A consisting of elements at indices i to j inclusive
$A[i][j]$ or $A[i, j]$	the element in i-th row and j-th column of two-dimensional array A
$A[i_1 : i_2][j_1 : j_2]$	2D subarray of two-dimensional array A consisting of elements from i_1-th to i_2-th rows and from j_1-th to j_2-th column, inclusive
$\binom{n}{k}$	binomial coefficient: number of ways of choosing k elements from a set of n items
$n!$	n-factorial, the product of the integers from 1 to n, inclusive
$O(f(n))$	big-oh complexity of $f(n)$, asymptotic upper bound
$\Theta(f(n))$	big-theta complexity of $f(n)$, asymptotically tight bound
$\Omega(f(n))$	big-Omega complexity of $f(n)$, asymptotic lower bound
$x \bmod y$	mod function
$x \oplus y$	bitwise-XOR function
$x \approx y$	x is approximately equal to y
`null`	pointer value reserved for indicating that the pointer does not refer to a valid address
$\emptyset$	empty set
∞	infinity: Informally, a number larger than any number. Rigorously, a set is infinite iff it can be mapped one-to-one to a proper subset of itself.
$\mathcal{Z}$	the set of integers $\{\ldots, -2, -1, 0, 1, 2, 3, \ldots\}$
$\mathcal{Z}^+$	the set of nonnegative integers $\{0, 1, 2, 3, \ldots\}$
$\mathcal{Z}_n$	the set $\{0, 1, 2, 3, \ldots, n-1\}$
$\mathcal{R}$	the set of real numbers
$\mathcal{R}^+$	the set of nonnegative real numbers
$x \ll y$	much less than
$x \gg y$	much greater than
$A \mapsto B$	function mapping from domain A to range B
$\Rightarrow$	logical implication
iff	if and only if
$\Pr(\mathcal{E})$	probability of event $\mathcal{E}$

Index of Terms

2D array, 37, 57–59, 88, 120, 122–124, 132, 137, 167, 170, 197–201, 268, 269, 345, 348, 356, 363, 364, 374, 380, 386, 395, 419, 456
2D subarray, 57, 120, 124, 197, 345–348
2∃-connected, 133, 134, 378
2∀-connected, 133, 134, 378
2∀-connectedness, 379
$O(1)$ space, 24, 25, 45, 53, 59, 76, 78, 90, 98, 109, 118, 183, 195, 202, 205, 211–213, 245, 246, 261, 273, 274, 281, 306, 320, 340, 428
0-1 knapsack problem, 140, 389, 390

abstract analysis patterns, 22, **37**
abstract data type, *see* ADT
acquired immune deficiency syndrome, *see* AIDS
adjacency list, 131, **131**, 421
adjacency matrix, 131, **131**
ADT, 24, **24**, 25, 26, 67, 70, 72
AIDS, **424**
AKS primality testing, 45
algorithm design patterns, 22, **28**
all pairs shortest paths, 33, 135, 380, 386
  Floyd-Warshall algorithm for, 386
alternating sequence, **342**
amortized, 67, 232, 290, 344
amortized analysis, 52, 92, 231
API, **25**, 26, 71, 81, 110, 152, 153, 229, 250, 325, 421
application programming interface, *see* API
approximation, **28**, 35, 36, 114, 150, 155
approximation algorithm, 138, 394, 425
arbitrage, 41, **41**, 137, **137**, 161, 162, 387, **387**, 388, 424, 444, 445
array, 1–4, 11, 13, 23, **23**, 24, 25, 29, 31, 34–36, 40, 44, 45, 50, 52, **52**, 53–56, 58, 60, 66, 67, 70–72, 81–84, 86–92, 97–101, 103–105, 107, 110, 115–121, 126, 140, 142, 145, 147, 156, 157, 166, 173, 174, 178, 183, 184, 186–189, 192, 193, 198, 199, 204, 205, 229, 234, 249, 251, 254, 255, 258–266, 270–275, 282, 287, 289, 292, 293, 295, 297, 298, 303–305, 314, 315, 323, 324, 334, 339–344, 349, 358, 363, 390, 391, 398, 403, 409, 419, 427, 431, 432, 452–454
  2D, *see* 2D array
  bit, *see* bit array
  deletion from, 52
articulation point, **379**, 380
ascending sequence, **342**
augmenting path, 463, **463**
auxiliary elements, 43
AVL tree, 27, 309

back edge, 379
backtracking, 114, 395, 401
balanced BST, 27, 80
  height of, 309
balanced tree, 236
  height of, 236
balls and bins, **158**
BCC, **379**, 380
Bellman-Ford algorithm, 42, 388
  for negative-weight cycle detection, 388
Bernoulli random variable, 155, 434
BFS, 132, **132**, 307, 338, 374–376, 380, 413
BFS tree, 374, 376, 377, 413
biconnected component, *see* BCC
binary search, 3, 4, 13, 20, 33, 34, 84, **84**, 85, 86, 98, 105, 115, 138, 260, 263–267, 295, 342, 343, 365, 366, 417
binary search tree, 3, 23, **23**, 25, *see* BST, 92
  AVL tree, 27, 309
  deletion from, 23, 27
  height of, 104, 107, 109, 309, 314, 320–322, 332
  red-black tree, 27, 104, 309
binary tree, *see also* binary search tree, **23**, 25, 26, 70, 71, 73–78, 80, 94, 104, 109, 131, 228, 235–245, 269, 278, 305, 320, 368, 370, 464

complete, 74, 80, 367
full, 74
height of, 23, 26, 27, 74–76, 78, 235, 236, 238, 245
perfect, 74
binomial coefficient, 42, 122, 353
bipartite graph, 135, 377, 460–462, 464
complete, 464
bipartite matching, 33, 460, 463
bit array, 20, 175, 197, 272, 279, 286
bitonic sequence, 342, **342**
Bloom filter, 23, **286**, 287
Boost, 12
Boyer-Moore algorithm, 40, 203
branch and bound, 138, 198, 393, 400
breadth-first search, *see* BFS
bridge, **379**, 380
brute-force solution, 39
BST, 12, 26, **26**, 27, 28, 31, 53, 68, 81, 104–110, 112, 113, 223, 234, 260, 292, 297, 303, 305–309, 311, 312, 314–322, 325–327, 330–332
buffer, 147, 292, 409, 419
busy wait, 408

caching, **28**, 34
capacity constraint, 140, 465
cardinality, 79, 102, 103, 158, 302, 303, 403, 428, 436, 462, 463
Cascading Style Sheets, *see* CSS
case analysis, 37, **38**
categorical starvation, 409, **409**
central processing unit, *see* CPU
CGI, **427**
Chebyshev inequality, 434
chessboard, 30, 43, 93, 168, 277, 398, 460, 461
mutilated, 30, 461
child, 68, 71, 74, 77, 104, 131, 238–240, 242, 243, 247, 278, 308, 321, 368, 371, 379, 413
left, *see* left child
right, *see* right child
circular queue, *see also* queue
clause, 143, 401
clique, **133**, 457
closed interval, 27, 102, 111, 113, 300, 301, 332
CNF, 143, **143**, 401
CNF-SAT, 138, **138**
code
Elias gamma, 50, 178
hash, *see* hash code
Huffman, 127, 368–371
coin changing, 125
Collatz conjecture, 143, **143**, 148, 401, 410, 411
coloring, 42, 58, 129, 373
diverse coloring, 129, 373, 374
two-coloring, 129, 373
column constraint, 197
combination, 31, 122, 354, 355
Commons Gateway Interface, *see* CGI
complete binary tree, 74, **74**, 80, 367
height of, 74, 367
complete bipartite graph, 464
complex number, 47, 111, 326
complexity analysis, **44**
concurrency, 4, 410
conjunctive normal form, *see* CNF
conjunctive normal form satisfiability, *see* CNF-SAT
connected component, 27, 131, **131**, 376, 381
connected directed graph, **131**
connected graph, 133, **133**
connected undirected graph, 131, **131**, 135
connected vertices, 131, **131**
constraint, 1, 25, 32, 42, 76, 101, 119, 126, 128, 135, 141, 142, 146, 147, 149, 166, 167, 225, 226, 238, 305, 358, 372, 373, 382, 391, 393, 409, 415, 443, 458, 466–468
capacity, 140, 465
column, 197
delay, 141, 392
design, 123, 124, 359
equality, 134, 381
flow, 466
hard, 419
inequality, 134, 381
placement, 100, 168, 296, 462
precedence, 403
row, 197
soft, 419
space, 36
stacking, 69, 224, 225
sub-grid, 197
subarray sum, 343
synchronization, 404
convex sequence, **342**
counting sort, 53, **53**, 292
CPU, **34**, 37
CSS, **427**
cumulative distribution function, 155

DAG, 123, **123**, 382, 462, 463, 465
data center, 134, 154, 425
data network, 134
data structure, **22**
data structure patterns, 22, **22**
database, 34, 125, 152, 420, 423, 426
deadlock, **145**, 410
decision tree, 423
decrease and conquer, **115**, 263, 270
degree

of a node in a rooted tree, 338
of a polynomial, 45, 304, 464
of a subtree, 338
delay constraint, 141, 392
deletion
from arrays, 52
from binary search trees, 23, 27
from doubly linked lists, 70
from hash tables, 23, 92
from heaps, 23
from linked list, 23
from max-heaps, 80
from priority queues, 26
from queues, 23, 71
from singly linked lists, 65
from stacks, 23, 71
dense graph, **380**
depth
of a function call stack, 292
of a node in a binary search tree, 307, 322
of a node in a binary tree, 23, 74, **74**, 245, 246
of a node in a Huffman tree, 371
of the function call stack, 45, 176
depth-first search, 45, *see* DFS
deque, **70**, 233
dequeue, 70
design constraint, 123, 124, 359
determinant, 445
DFS, 132, **132**, 374, 378–380, 383, 389
diameter
of a network, 413
of a tree, 116, 337, 338
Dijkstra's algorithm, 4, 27, 384–386
implemented with a Fibonacci heap, 27
directed acyclic graph, *see* DAG
directed graph, 130, *see also* directed acyclic graph, *see also* graph, **130**, 131, 133, 135, 167, 383, 412, 466
connected directed graph, 131
tournament, 167
weakly connected graph, 131
weighted, 387
discovery time, **132**, 379
disjoint-set data structure, **27**, 28, 190
distance
Euclidean, 142
Levenshtein, 37, 120, 121, 145, 151, 349–351, 403, 417
distributed memory, **144**, 145
distribution
normal, 443
of the elements, 36
of the inputs, 45
of the numbers, 34
Pareto, 153, 422
diverse coloring, 129, **129**, 373, **373**, 374
divide and conquer, 2, 3, 12, **28**, 29–31, 40, 114, 115, 117, 254, 332, 335, 338, 400
divisor, 50, 446
greatest common divisor, 50
DNS, **426**
Document Object Model, *see* DOM
DOM, 427, **427**
Domain Name Server, *see* DNS
double-ended queue, *see* deque
doubly linked list, 23, *see also* linked list, 24, 62, **62**, 70, 107, 208, 288, 316, 470
deletion from, 70
DP, 12, 31, **31**, 34, 42, 117, 118, 125, 138, 351, 354–358, 366, 389, 390, 393, 432, 439, 440, 454, 464
dynamic order statistics, 23
dynamic programming, 3, *see* DP, **31**, 117

edge, 33, 41, 42, 116, 130, **130**, 131, 133, 135, 136, 371, 376–382, 384, 387, 393, 403, 412, 424, 460, 461, 470
bridge, 379, 380
efficient frontier, 31, **31**, 34, 37, 343, 344
Elias gamma code, 50, **50**, 178
elimination, **28**, 33, 84
enqueue, 70
equality constraint, 134, 381
equivalence class, 55, **55**, 189
equivalence relation, 55, **55**, 188
Ethernet, 116
Euclidean distance, 142
Euclidean space, 425
expected value, 155, **155**, 158, 188, 436, 437, 446
Extensible Markup Language, *see* XML
extract-max, 80, 250
extract-min, 249, 256, 257, 368, 422

fast Fourier Transform, *see* FFT
FFT, **304**
Fibonacci heap, 27
in Dijkstra's algorithm, 27
Fibonacci number, 117
finishing time, **132**, 383
first-in, first-out, 25, *see also* queue, 70, 71
flow constraint, 466
flow network, 460, 465
Floyd-Warshall algorithm, 386
fractional knapsack problem, 390
free tree, 131, **131**
full binary tree, 74, **74**
function
hash, *see* hash function
probability density, 155, 438

recursive, 29

garbage collection, 144
lazy, 290
Gaussian elimination, 455
Gaussian integer, 111, **111**, 326
Gaussian prime, 111, **111**, 326
Gaussian random variable, 155, 156
GCD, 50, **50**, 179, 195, 196
generalization principle, 30
Global Positioning System, *see* GPS
global variable, 236, 237, 372
GPS, **413**
graph, *see also* undirected graph, 33, 41, 42, 45, 114, 128–130, *see also* directed graph, **130**, 131, *see also* tree, 133, 134, 373, 377, 380, *see also* flow network
bipartite, 135, 377, 460–462, 464
coloring, 129
complete bipartite, 464
tour of, 141
graph modeling, **38**, 41, 132
graphical user interfaces, *see* GUI
greatest common divisor, *see* GCD
greedy, 19, **28**, 32, **32**, 34, 36, 114, 124–126, 363, 365, 395
GUI, **144**, 152

Hamiltonian cycle, 138, 458, 461
Hamiltonian path, 167, 167, 457, 457, 458
hard constraint, 419
hash code, 34, 35, 53, 92, **92**, 93, 126, 276–278, 283, 286, 287, 365, 415, 420–422
hash function, 23, 26, **26**, 33, 35, 92, 93, 126, 263, 276–280, 283, 286, 287, 415, 422
hash table, 4, 20, 23, **23**, 26, 28, 31, 53, 59, 90, 92, 95, 208, 235, 247, 262, 279–282, 284, 286–290, 297, 298, 325, 326, 330, 351, 375, 380, 402, 407, 417, 421, 431
deletion from, 23, 92
head
of a biased coin, 158, 434
of a deque, 70
of a linked list, 62–64, 207–209, 211, 213, 214
of a mole, 166
of a postings list, 66, 217
of a queue, 70, 228–230, 290, 326, 419
heap, 22, 23, **23**, 26, 27, 80, **80**, 81, 98, 117, 250, 254, 255, 422
Fibonacci heap, 27
max-heap, 80, 98
min-heap, 80, 98
priority queue, 26
treap, 27

heapsort, 98
height
of a balanced BST, 309
of a balanced tree, 236
of a binary search tree, 104, 107, 109, 309, 314, 320–322, 332
of a binary tree, 23, 26, 27, 74, **74**, 75, 76, 78, 235, 236, 238, 245
of a building, 69, 115, 120, 225, 332, 333, 344
of a complete binary tree, 74, 367
of a domino, 43
of a event rectangle, 101
of a line segment, 27, 111, 327, 329
of a perfect binary tree, 74
of a player, 100, 296
of a rectangle, 50
of a stack, 236
of a statue, 99, 293
height-balanced, 27
height-balanced BST, 107, 314
height-balanced tree, 27
highway network, 136, 386
HTML, **423**, 424, 425, 427
HTTP, **91**, 145, 155, 423, 426, 427
Huffman code, 127, 368–371
Huffman tree, 369, 370
HyperText Markup Language, *see* HTML
Hypertext Transfer Protocol, *see* HTTP

I/O, 19, 145, 405
IDE, **13**
IID, **434**, 436
in-place sort, **98**
incremental improvement, **28**, 32, 33
independent and identically distributed, *see* IID
indicator random variable, 271, 436
indirect sort, 293
inequality
Chebyshev, 434
linear, 138, 468
triangle, 393, 395, 437
inequality constraint, 134, 381
insertion sort, 292, 304
integral development environment, *see* IDE
International Organization for Standardization, *see* ISO
International Standard Book Number, *see* ISBN
Internet Protocol, *see* IP
interval tree, 23, 78, 332
intractability, 4, 138
invariant, **38**, 43, 44, 255, 256
inverted index, 99, 416
IP, 90, **90**, 91, 148, 151, 272, 412, 413, 417, 426, 427
ISBN, 97, **97**, 290

ISO, **97**
isomorphic binary trees, **94**
isomorphic tree, **168**
iterative refinement, **38**, 39, 40

JavaScript Object Notation, *see* JSON
JSON, **427**

knapsack problem
  0-1, 140, 389, 390
  fractional, 390
Knuth-Morris-Pratt algorithm, 203
Kruskal's algorithm, 27

Lagrangian relaxation, 393
LAN, **116**
last-in, first-out, 25, *see also* stack, 67, 71, 223
lazy garbage collection, 290
LCA, 78, **78**, 79, 109, 245–247, 320
leaf, 23, 74, 75, 78, 123, 243, 244, 247, 367–369, 371, 372, 412
Least Recently Used, *see* LRU
left child, 68, 70, 71, 73, 74, 77, 112, 131, 235, 239, 240, 242–245, 258, 305, 307, 308, 311, 315, 331, 332, 367, 368
left subtree, 27, 73–77, 104, 106, 223, 237, 240, 241, 243, 244, 305, 307, 308, 311, 313, 315, 317, 319
length
  of a string, 287
Levenshtein distance, 37, 120, **121**, 145, 151, **349**, **350**, 351, 403, 417
line segment, 27, 111, 159, 327, 329, 437, 438
  height of, 27, 111, 327, 329
linear equation, 455
linear inequality, 138, 468
linear programming, 33, 45, 460, 467
  simplex algorithm for, 33, 45, 467
linear search, 265
linked list, 23, 25, 470
list, **23**, *see also* singly linked list, 63–67, 70, 78, 92, 107, 207, 208, 210–216, 243, 256, 292, 314, 316, 318
  postings, 66, 68, 217, 218, 223
livelock, **145**, 410
load
  of a hash table, 92
  of a server, 126, 365
local area network, *see* LAN
lock
  deadlock, 145, 410
  livelock, 145, 410
longest alternating subsequence, **342**
longest bitonic subsequence, **342**
longest convex subsequence, **342**
longest nondecreasing subsequence, 31, 119, **119**, 340, **340**
longest path, 116, 338, 371, 382
longest weakly alternating subsequence, **342**
lowest common ancestor, *see* LCA, 109
LRU, **97**, 290

Markowitz bullet, 331
matching, **135**
  bipartite, 33, 460, 463
  maximum, 135, 461, 462
  maximum weighted, 135
  weighted bipartite, 464
matrix, 131, 152, 169, 466
  adjacency, 131
  multiplication of, 144, 421
  payoff, 170, 467
matrix multiplication, 144, 421
max-heap, 80, 83, 98, 250–252, 254, 255, 258
  deletion from, 23, 80
maximum bipartite matching, *see* bipartite matching
maximum flow, 33, 135, **135**, 169, 464, 465
maximum matching, 135, 461, 462
maximum weighted matching, **135**
mean, 156, 162, 434, 443, 445
median, 35, 39, 40, 63, 82, 210, 254–256, 290, 335, 336, 449, 450
merge sort, 98, 114, 249, 292, 323, 334
min-heap, 23, 26, 34, 80, 98, 248, 250, 253–257, 271, 272, 292, 407, 421
  in Huffman's algorithm, 368
minimum spanning tree, 28, *see* MST, 135, **135**, 137, 394
  Kruskal's algorithm for, 27
Morris traversal, **75**, 76
MST, 33, **114**, 115, 137, 389, 393, 394
multicore, 144, 148
mutex, 146, 147, 409, 410
mutilated chessboard, 30, 461

negative-weight cycle, 388
network, 144, 148, 413, 426
  data, 134
  highway, 136, 386
  local are network, 116
  network bandwidth, 34, 91
  network layer, 415
  network route, 82
  network session, 157
  network traffic control, 72
  social, 94, 134, 149, 380
network bandwidth, 34, 91
network layer, 415
network session, 157

node, 116, 338
nondecreasing subsequence, 341
normal distribution, 443
NP, **138**
NP-complete, 36, 54, 142, 143, 383, 389, 393, 403
NP-hard, 125, 141, 142
null string, 371

open interval, 300
operating system, *see* OS
order statistics, 89, **89**
    dynamic, 23
ordered pair, 141, 283
ordered tree, 131, **131**
OS, **4**, 150, 419, 423
overflow
    integer, 42, 85, 122, 273, 353, 354, 402
    of a stack, 231
overlapping intervals, 299

palindrome, 44, 66, 95, **95**, 166, 216, 280, 281, 351, 450, 451
palindromic string, 166
parallel algorithm, 139
parallelism, 28, 34, 144, 145
parent-child relationship, 74, 131, 168
Pareto distribution, 153, **153**, 422
partition, 34, **55**, 95, 101, 115, 126, 142, 190, 280, 287, 335, 354, 391, 402, **414**, 415, 421, 422
path, **130**
    augmenting, 463
    Hamiltonian, 167, 457
    shortest, *see* shortest paths
payoff matrix, 170, **170**, 467
PDF, **9**, 419
perfect binary tree, 74, **74**
    height of, 74
permutation, 55, 56, **56**, 89, 156, 157, 189, 192–195, 293, 355, 427, 428, 431, 436
    random, 156, 158, 188
    uniformly random, 156, 158, 427, 429, 436
placement constraint, 100, 168, 296, 462
Poisson random variable, 155
Polish notation, 222
Portable Document Format, *see* PDF
postings list, 66, **66**, 68, 217, 218, 223
power set, 48, **48**, 175
precedence constraint, 403
prefix
    of a sequence, 205
    of a string, 79, 127, 151, 247, 248, 371, 417
prefix sum, 40, 187, 265, 343
primality, *see* prime
prime, 45, 50, **50**, 104, 180, 277, 326
    Gaussian, 111, 326
priority queue, 26, **26**
    deletion from, 26
probability density function, 155, **155**, 438
probability distribution function, 188
production sequence, 133, 375

queue, 23, 25, 26, 70, **70**, 71, 72, 81, 147, 199, 200, 228–235, 250, 290, 307, 325, 326, 374, 375, 415, 419, 422
    deletion from, 71
    priority, 26
quicksort, 3, 24, 45, 52, 98, 114, 118, 155, 255, 292

Rabin-Karp algorithm, 203
race, 145, 404, 409
radix sort, 98, 297
RAM, **34**, 80, 81, 90, 96, 97, 110, 151, 248, 251, 272, 286, 287, 325, 416, 417, 421
random access memory, *see* RAM
random number generator, 156, 157, 426–430
random permutation, 156, 158, 188
    uniformly, 156, 158, 427, 429, 436
random variable, 155, **155**, 160, 162, 366, 434–436, 442, 445
    Bernoulli, 155, 434
    Gaussian, 155, 156
    indicator, 271, 436
    Poisson, 155
    uniform, 155
randomization, 28, **28**, 35, 92, 114
randomized algorithm, 45, 161, 443
reachable, **130**, 132
recursion, 12, **28**, 29, **29**, 30, 31, 42, 62, 68, 71, 76, 117, 179, 200, 205, 206, 215, 223, 226, 228, 243, 336, 348, 354, 355, 391, 412, 432
recursive function, 29, **29**
red-black tree, 27, **104**, 309
reduction, **38**, 41, 81, 114
regular expression, 29, 60, **60**, 207, 351
rehashing, 53, 92
Reverse Polish notation, 25, *see* RPN
right child, 68, 70, 71, 73, 74, 77, 112, 131, 223, 235, 236, 239, 242, 243, 245, 258, 305, 307–309, 311, 315, 331, 332
right subtree, 27, 73–76, 104, 106, 223, 237, 240, 241, 243, 244, 305, 308, 309, 313, 315, 317, 319, 332
RLE, 59, **59**
rolling hash, **203**, 276
root, 70, 71, 73–79, 94, 104–106, 108, 124, 127, 131, 148, 151, 223, 228, 235–247, 251, 253, 258, 305, 307, 308, 311, 313–315,

317–320, 331, 332, 359, 367–369, 371, 372, 379, 393, 412
rooted tree, 123, 124, 127, 131, **131**, 148, 168, 359, 412, 463
router, 151, 417
row constraint, 197
RPN, 68, **68**, 221
run-length encoding, *see* RLE

scheduling, 101, 136, 363, 383, 403
searching
binary search, *see* binary search
linear search, 265
sequence, 342, 350
alternating, 342
ascending, 342
bitonic, 342
convex, 342
production, 133, 375
weakly alternating, 342
shared memory, 144, **144**
Short Message Service, *see* SMS
shortest path, 33, 132, 136, 138, 338, 374, 380, 384–386, 393, 424
Dijkstra's algorithm for, 4, 27, 384–386
shortest path, unweighted case, 374
shortest paths, **135**
shortest paths, unweighted edges, 375
sibling, 371
signature, 35, 420
simplex algorithm, 33, 45, 467
singly linked list, 23, 25, 62, **62**, 63–65, 213, 214
deletion from, 65
skip list, 27
small example, 38, **38**, 39, 446
SMS, **146**
social network, 13, 94, 134, 149, 380
soft constraint, 419
sorted doubly linked list, 107, 108, 315
sorting, 28, 29, 34, 35, 39, 45, 52, 69, 81, 86, 89, 98, 99, 103, 114, 249, 251, 265, 281, 292, 293, 296–300, 304, 336
counting sort, 53, 292
heapsort, 98
in-place, 98, 292, 298
in-place sort, 98
indirect sort, 293
insertion sort, 292, 304
merge sort, 98, 114, 249, 292, 323, 334
quicksort, 24, 45, 52, 98, 114, 118, 155, 255, 292
radix sort, 98, 297
stable, 98, 292, 297
stable sort, 98
space
Euclidean, 425
space complexity, **2**
space constraint, 36
spanning tree, 131, *see also* minimum spanning tree
SQL, **16**, 125
square root, 33, 45, 88
stable sort, **98**, 271
stack, 23, 25, 31, 67, **67**, 69, 71, 76, 81, 103, 200, 215, 219–223, 226, 227, 231, 232, 242, 243, 250, 344, 374
deletion from, 71
height of, 236
stacking constraint, 69, 224, 225
Standard Template Library, *see* STL
starvation, **145**, 147
categorical, 409
starvation-free, 410
thread, 409
state, **28**
STL, **104**, 330, 398
streaming
algorithm, 45, 288
fashion input, 81, 82, 97, 287
string, 23, **23**, 26, 28, 29, 37, 40, 41, 44, 49, 50, 55, 56, 59–61, 68, 69, 79, 93, 95–97, 100, 112, 120, 121, 127, 133, 142, 145–147, 149, 151, 166, 176–178, 190, 191, 201–206, 221, 227, 247, 248, 276, 279–282, 284, 286, 287, 297, 330, 349–352, 368, 375, 395, 403, 404, 407, 409, 414, 416–419, 450, 451
null, 371
palindromic, 166
string matching, 29, 37, 59, 119
Boyer-Moore algorithm for, 40, 203
Knuth-Morris-Pratt algorithm for, 203
Rabin-Karp algorithm for, 203
strongly connected directed graph, **131**
Structured Query Language, *see* SQL
sub-grid constraint, 197
subarray, 2, 35, 40, 52, 55, 97, 103, 117–119, 183, 184, 187, 188, 200, 201, 234, 249, 254, 259, 270, 287–289, 314, 339, 340, 342–344, 428
2D, *see* 2D subarray
subarray sum constraint, 343
subsequence, 27, 31, 141, 318, 319, 341, 350
longest alternating, 342
longest bitonic, 342
longest convex, 342
longest nondecreasing, 31, 119, 340
longest weakly alternating, 342
nondecreasing, 341
subset sum, 54, **54**, 389

substring, 59, 61, 96, 203, 205, 206, 227, 286, 287, 351, 420
subtree, 74, 77, 94, 106, 107, 113, 168, 235–238, 240, 279, 305, 307, 309, 314, 321, 331, 332, 338, 368, 372, 412, 464
  left, *see* left subtree
  right, *see* right subtree
Sudoku, 57, **57**, 142, 197, 198, 395
suffix, 61
synchronization constraint, 404

tail
  of a biased coin, 158, 434, 435
  of a deque, 70
  of a linked list, 62, 65, 208, 211, 214
  of a queue, 70, 229, 230, 250, 290, 419
tail recursion, 115, 292
tail recursive, 117, 215, 312
TCP, **148**, 427
thread starvation, 409
time complexity, **2**, 11
timestamp, 26, 82, 110, 326, 424
topological order, 383, 403
topological ordering, 382, 383
total order, 82, 335
tour, 33, 141, **141**, 393, 394
tournament, 167, **167**, 457, 458
tournament tree, 366–368
Transmission Control Protocol, *see* TCP
treap, 27, **27**
tree, 131, **131**, 132, 133, 393
  AVL, 27, 309
  BFS, 374, 376, 377
  binary, *see* binary tree
  binary search, *see* binary search tree
  decision, 423
  diameter, 116, 337, 338
  free, 131
  Huffman, 369, 370
  interval, 23, 78, 332
  isomorphic, 168
  ordered, 131
  red-black, 27, 104, 309
  rooted, 123, 124, 127, 131, 148, 168, 359, 412, 463
  tournament, 366–368
  treap, 27
triangle inequality, 393, 395, 437
trie, 28, **28**, **247**, 417
triomino, 30, 461
two-coloring, 129, 373
two-dimensional tree, 124, 359, 362

UI, **19**, 144, 425
unconnected graph, 134
undirected complete graph, **167**
undirected graph, 27, 42, 129, **130**, 131, 133–135, 137, 373–379, 381, 389, 457
  clique, 133
  undirected complete graph, 167
  weighted, 114
uniform random variable, 155
Uniform Resource Locators, *see* URL
uniformly random permutation, 156, 158, 427, 429, 436
UNIX, 152, 414, 419, 426
URL, 9, **121**, 149, 154, 421, 426, 427
user interface, *see* UI

variance, 155, 156, 434, 443
variation, **38**, 42
vertex, 33, 41, 42, 114, 115, 129, 130, **130**, 131–133, 135, 136, 141, 167, 181, 372–383, 386, 388, 392, 393, 403, 412, 421, 424, 425, 457, 458, 460–466, 470
  articulation point, 379, 380
  black vertex in DFS, 378
  connected, 131
  gray vertex in DFS, 378
  white vertex in DFS, 378

weakly alternating sequence, **342**
weakly connected, 412
weakly connected graph, **131**
weighted bipartite matching, 464
weighted directed graph, **387**
weighted undirected graph, 114, 115
work-queue, 411
World Wide Web, 154, 426
write an equation, **38**, 42

XML, **427**

Acknowledgments

Several of our friends, colleagues, and readers gave feedback. We would like to thank Senthil Chellappan, Yi-Ting Chen, Monica Farkash, Cheng-Yi He, Dongbo Hu, Jing-Tang Keith Jang, Matthieu Jeanson, Gerson Kurz, Hari Mony, Shaun Phillips, Gayatri Ramachandran, Ulises Reyes, Kumud Sanwal, Tom Shiple, Ian Varley, Shaohua Wan, Don Wong, Xiang Wu, and Chih-Chiang Yu for their input.

I, Adnan Aziz, thank my teachers, friends, and students from IIT Kanpur, UC Berkeley, and UT Austin for having nurtured my passion for programming. I especially thank my friends Vineet Gupta, Tom Shiple, and Vigyan Singhal, and my teachers Robert Solovay, Robert Brayton, Richard Karp, Raimund Seidel, and Somenath Biswas, for all that they taught me. My coauthor, Amit Prakash, has been a wonderful collaborator for many years—this book is a testament to his intellect, creativity, and enthusiasm. My coauthor, Tsung-Hsien Lee, brought a passion that was infectious and inspirational; I look forward to a lifelong collaboration with him.

I, Amit Prakash, have my coauthor and mentor, Adnan Aziz, to thank the most for this book. To a great extent, my problem solving skills have been shaped by Adnan. There have been occasions in life when I would not have made it through without his help. He is also the best possible collaborator I can think of for any intellectual endeavor. I have come to know Tsung-Hsien through working on this book. He has been a great coauthor. His passion and commitment to excellence can be seen everywhere in this book. Over the years, I have been fortunate to have had great teachers at IIT Kanpur and UT Austin. I would especially like to thank my teachers Scott Nettles, Vijaya Ramachandran, and Gustavo de Veciana. I would also like to thank my friends and colleagues at Google, Microsoft, IIT Kanpur, and UT Austin for many stimulating conversations and problem solving sessions. Finally, and most importantly, I want to thank my family who have been a constant source of support, excitement, and joy all my life and especially during the process of writing this book.

I, Tsung-Hsien Lee, would like to thank my coauthors, Adnan Aziz and Amit Prakash, who give me this once-in-a-life-time opportunity. I also thank my teachers Wen-Lian Hsu, Ren-Song Tsay, Biing-Feng Wang, and Ting-Chi Wang for having initiated and nurtured my passion for computer science in general, and algorithms in particular. I would like to thank my friends Cheng-Yi He, Da-Cheng Juan, Chien-Hsin Lin, and Chih-Chiang Yu, who accompanied me on the road of savoring the joy of programming contests; and Kuan-Chieh Chen, Chun-Cheng Chou, Ray Chuang, Wen-Sao Hong, Wei-Lun Hung, Nigel Liang, Huan-Kai Peng, and Yu-En Tsai, who give me valuable feedback on this book. Last, I would like to thank all my friends and colleagues at Facebook, National Tsing Hua University, and UT Austin for the brain-storming on puzzles; it is indeed my greatest honor to have known all of you.

Adnan Aziz	*Austin, Texas*
Amit Prakash	*Belmont, California*
Tsung-Hsien Lee	*Mountain View, California*
October, 2012	