Elements of Programming Interviews

The Insiders' Guide

Adnan Aziz

Tsung-Hsien Lee

Amit Prakash

This document is a sampling of our book, Elements of Programming Interviews (EPI). Its purpose is to provide examples of EPI's organization, content, style, topics, and quality.

We'd love to hear from you—we're especially interested in your suggestions as to where the exposition can be improved, as well as any insights into interviewing trends you may have.

You can buy EPI with at Amazon.com.

---

http://ElementsOfProgrammingInterviews.com

Adnan Aziz is a professor at the Department of Electrical and Computer Engineering at The University of Texas at Austin, where he conducts research and teaches classes in applied algorithms. He received his Ph.D. from The University of California at Berkeley; his undergraduate degree is from Indian Institutes of Technology Kanpur. He has worked at Google, Qualcomm, IBM, and several software startups. When not designing algorithms, he plays with his children, Laila, Imran, and Omar.

Tsung-Hsien Lee is a Software Engineer at Google. Previously, he worked as a Software Engineer Intern at Facebook. He received both his M.S. and undergraduate degrees from National Tsing Hua University. He has a passion for designing and implementing algorithms. He likes to apply algorithms to every aspect of his life. He takes special pride in helping to organize Google Code Jam 2014.

Amit Prakash is a co-founder and CTO of ThoughtSpot, a Silicon Valley startup. Previously, he was a Member of the Technical Staff at Google, where he worked primarily on machine learning problems that arise in the context of online advertising. Before that he worked at Microsoft in the web search team. He received his Ph.D. from The University of Texas at Austin; his undergraduate degree is from Indian Institutes of Technology Kanpur. When he is not improving business intelligence, he indulges in his passion for puzzles, movies, travel, and adventures with Nidhi and Aanya.

Table of Contents

Introduction

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 suffices to consider S. You may be tempted to simply return the difference of the
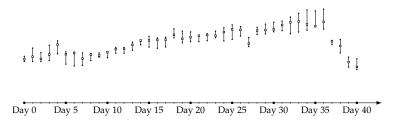
Figure 1: Share price as a function of time.

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 6.2 on Page 110.

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

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 8.

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.

The problem chapters are organized as follows. Chapters 5–15 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 16–19 cover advanced algorithm design principles, such as dynamic programming and heuristics, as well as graphs. Chapters 20–21 focus on distributed and parallel programming, and design problems. Each chapter begins with a summary of key concepts, followed by problems. Broadly speaking, problems are ordered by subtopic, with more commonly asked problems appearing first.

The notation, specifically the symbols we use for describing algorithms, e.g., $\sum_{i=0}^{n-1} i^2, [a, b), \langle 2, 3, 5, 7 \rangle, A[i : j], \Rightarrow, |S|, \{x \mid x^2 > 2\}$, etc., is summarized starting on Page 179. It should be familiar to anyone with a technical undergraduate degree, but we still request you to review it carefully before getting into the book, and whenever you have doubts about the meaning of a symbol. Terms, e.g., BFS and dequeue, are indexed starting on Page 181.

Problems, solutions, variants, ninjas, and hints

Most solutions in EPI are based on basic concepts, such as arrays, hash tables, and binary search, used in clever ways. Some solutions use relatively advanced machin-

ery, e.g., Dijkstra's shortest path algorithm. You will encounter such problems in an interview only if you have a graduate degree or claim specialized knowledge.

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 103. A guide to reading C++ programs for Java developers is given on Page 103. Source code, which includes randomized and directed test cases, can be found at the book website. Java equivalents for all C++ programs are available at the same site. System design problems are conceptual and not meant to be coded; a few algorithm design problems are also in this spirit.

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.

Approximately a fifth 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. Questions without a ninja label should be solvable within an hour-long interview and, in some cases, take substantially less time.

Often, your interviewer will give you a brief suggestion on how to proceed with a problem if you get stuck. We provide hints in this style on Page 99.

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. Some of the material in the later chapters, specifically dynamic programming, graphs, and greedy algorithms, 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. *Algorithms* by Dasgupta, *et al.* is succinct and beautifully written; *Introduction to Algorithms* by Cormen, *et al.* is an amazing 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.

EPI Sampler

This document is a sampling of EPI. Its purpose is to provide examples of EPI's organization, content, style, topics, and quality. You can get a better sense of the problems not included in this document by visiting the EPI website.

We have had to make small changes to account for the sampling process. For example, the patterns chapter in this document does not refer to problems that illustrate the pattern being discussed. Similarly, we do not include the study guide from EPI, which specifies the problems to focus on based on the amount of time you have to prepare.

This document is automatically built from the source code for EPI. There may be abrupt changes in topic and peculiarities with respect to spacing because of the build process.

Reader engagement

Many of the best ideas in EPI came from readers like you. The study guide, ninja notation, and hints, are a few examples of many improvements that were brought about by our readers.

The companion website, ElementsOfProgrammingInterviews.com, includes a Stack Overflow-style discussion forum, and links to our social media presence. It also has links blog postings, code, and bug reports.

Please fill out the EPI registration form, bit.ly/epireg, to get updates on interviewing trends, good practice problems, links to our blog posts, and advance information on the EPI roadmap. (You can always communicate with us directly—our contact information is on the website.)

Part I

The Interview

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 following page

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 next 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 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.

Table 1.1: A summary of nontechnical aspects of interviewing

The Interview Lifecycle, on the previous page	**At the Interview, on Page 11**
− Identify companies, contacts	− Don't solve the wrong problem
− Résumé preparation	− Get specs & requirements
◇ Basic principles	− Construct sample input/output
◇ Website with links to projects	− Work on concrete examples first
◇ LinkedIn profile & recommendations	− Spell out the brute-force solution
− Résumé submission	− Think out loud
− Mock interview practice	− Apply patterns
− Phone/campus screening	− Test for corner-cases
− On-site interview	− Use proper syntax
− Negotiating an offer	− Manage the whiteboard
	− Be aware of memory management
	− Get function signatures right
General Advice, on Page 15	**Conducting an Interview, on Page 18**
− Know the company & interviewers	− Don't be indecisive
− Communicate clearly	− Create a brand ambassador
− Be passionate	− Coordinate with other interviewers
− Be honest	◇ know what to test on
− Stay positive	◇ look for patterns of mistakes
− Don't apologize	− Characteristics of a good problem:
− Be well-groomed	◇ no single point of failure
− Mind your body language	◇ has multiple solutions
− Leave perks and money out	◇ covers multiple areas
− Be ready for a stress interview	◇ is calibrated on colleagues
− Learn from bad outcomes	◇ does not require unnecessary domain knowledge
− Negotiate the best offer	− 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

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. 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. If you cannot find a friend, you can still go through the same process, recording yourself.

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 integrated development environments (IDEs). 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.

Feel free to ask the interviewer what time and space complexity he would like in your solution. If you are told to implement an $O(n)$ algorithm or use $O(1)$ space, it can simplify your life considerably. It is possible that he will refuse to specify these, or be vague about complexity requirements, but there is no harm in asking.

Work on concrete examples: Consider Problem 5.5 on Page 44, 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 17.4 on Page 82), 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 to 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 IDE 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. We use the Google coding style standards in this book, and advise you to become proficient with them. They are available at code.google.com/p/google-styleguide

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 altogether. 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 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 un-interested, 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 com-pany. 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.

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), input/output (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 6.2 on Page 47 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 12.4 on Page 67 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 20.2 on Page 94 tests candidates on concurrency as well as data structures.

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, it 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.

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 the following categories:

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

These patterns are summarized in Table 4.1 on the facing page, Table 4.2 on Page 28, Table 4.3 on Page 33, and Table 4.5 on Page 37, respectively. Keep in mind that you may have to use more than one of these patterns for a given problem.

The two most common criteria for comparing algorithms are runtime and memory usage. Much of the discussion around patterns uses the "big-Oh" notation to describe these more formally. We briefly review complexity analysis and intractable problems at the end of this chapter.

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

The data structures described in this chapter are the ones commonly used. Other data structures, such as skip lists, treaps, Fibonacci heaps, tries, and disjoint-set data structures, have more specialized applications.

Solutions often require a combination of data structures. For example, tracking the most visited pages on a website 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	Fast access for element at an index, slow lookups (unless sorted) and insertions. Be comfortable with notions of iteration, resizing, partitioning, merging, etc.
Strings	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 node fields, pointer implementation. Be familiar with notion of balance, and operations maintaining balance. Know how to augment a binary search tree, e.g., interval trees and dynamic k-th largest.

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 integer width 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 $y = x \& {\sim}(x-1)$, where $\&$ is the bitwise AND operator and $\sim$ is the bitwise complement operator. The variable y is 1 at exactly the lowest set bit of x; all other bits in y are 0. For example, if $x = (0110)_2$, then $y = (0010)_2$. This calculation is correct both for unsigned and two's-complement representations. Consequently, this bit may be removed from x by computing $x \oplus y$, where $\oplus$ is the bitwise-XOR function. 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 65536 entry integer-valued array P, such that $P[i]$ is the number of bits set to 1 in i. If x is 64 bits, the result can be computed by decomposing x into 4 disjoint 16-bit words, $h3, h2, h1$, and $h0$. The 16-bit words are computed using bitmasks and shifting, e.g., $h1$ is $(x \gg 16 \& (1111111111111111)_2)$. The final result is $P[h3] + P[h2] + P[h1] + P[h0]$.

ARRAYS

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.

STRINGS

A string can be viewed as a special kind of array, namely one made out of characters. We treat strings separately from arrays because certain operations which are commonly applied to strings—for example, comparison, joining, splitting, searching for substrings, replacing one string by another, parsing, etc.—do not make sense for general arrays.

Our solution to the look-and-say problem illustrates operations on strings. The look-and-say sequence begins with 1; the subsequent integers describe the digits appearing in the previous number in the sequence. The first eight integers in the look-and-say sequence are $\langle 1, 11, 21, 1211, 111221, 312211, 13112221, 1113213211 \rangle$.
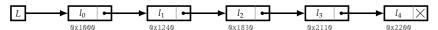
The look-and-say problem entails computing the n-th integer in this sequence. Although the problem is cast in terms of integers, the string representation is far more convenient for counting digits.

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 include repetitions. 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 also 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.

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, ×, 1, 2, +, +", "1, 1, +, −2, ×", 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.

BINARY TREES

A binary tree is a data structure that is used to represent hierarchical relationships. Binary trees are the subject of Chapter 10. 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 `isLocked()`, `lock()`, and `unLock()` methods. Naively 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.

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.

Let's say 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.

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 caveat 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 $O(n)$ complexity.

Suppose you were asked to write a function which takes a string s as input, and returns true if the characters in s can be permuted to form a string that is palindromic, i.e., reads the same backwards as forwards. For example, your function should return true for "GATTAACAG", since "GATACATAG" is a permutation of this string and is palindromic. Working through examples, you should see that a string is palindromic

if and only if each character appears an even number of times, with possibly a single exception, since this allows for pairing characters in the first and second halves.

A hash table makes performing this test trivial. We build a hash table H whose keys are characters, and corresponding values are the number of occurrences for that character. The hash table H is created with a single pass over the string. After computing the number of occurrences, we iterate over the key-value pairs in H. If more than one character has an odd count, we return false; otherwise, we return true.

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.

BINARY SEARCH TREES

Binary search trees (BSTs) are used to store objects that are comparable. BSTs are the subject of Chapter 15. The underlying idea is to organize the objects in a binary tree in which the nodes satisfy the BST property on Page 74. 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.

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 on the following page. 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 program-ming	Compute solutions for smaller instances of a given problem and use these solutions to construct a solution to the problem. Cache for performance.
Greedy algorithms	Compute a solution in stages, making choices that are locally optimum at step; these choices are never undone.
Invariants	Identify an invariant and use it to rule out potential solutions that are suboptimal/dominated by other solutions.

SORTING

Certain problems become easier to understand, as well as solve, when the input is sorted. The solution to the calendar rendering problem 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, in all likelihood, lead to the correct criterion.

Sorting is not appropriate when an $O(n)$ (or better) algorithm is possible. Another good example of a problem where a total ordering is not required is the problem of rearranging elements in an array described on Page 35. 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 often appropriate when the input is expressed using recursive rules, such as a computer grammar. More generally, searching, enumeration, divide-and-conquer, and decomposing a complex problem into a set of similar smaller instances are all scenarios where recursion may be suitable.

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 lowercase 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_1r_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.

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 18; 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.
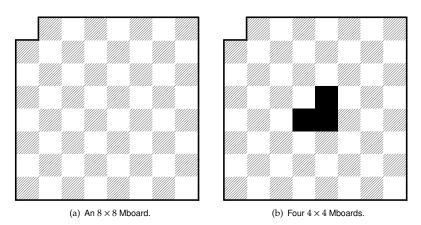

(a) An 8×8 Mboard. (b) Four 4×4 Mboards.

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. 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) on the previous page. 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 solving the number of pairs of elements in an array that are out of sorted order and computing the closest pair of points in a set of points in the plane.

DYNAMIC PROGRAMMING

Dynamic programming (DP) is applicable when the problem has the "optimal sub-structure" 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 the one ending with a 3 point play, and a combination ending with a 3 point play is different from the one ending with a 7 point play, etc.

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

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 17.1 on Page 155.

GREEDY ALGORITHMS

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. This algorithm 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.

Chapter 18 contains a number of problems whose solutions employ greedy algorithms. representative. Several problems in other chapters also use a greedy algorithm as a key subroutine.

INVARIANTS

One common approach to designing an efficient algorithm is to use invariants. Briefly, an invariant is a condition that is true during execution of a program. This condition may be on the values of the variables of the program, or on the control

logic. A well-chosen invariant can be used to rule out potential solutions that are suboptimal or dominated by other solutions.

An invariant can also be used to analyze a given algorithm, e.g., to prove its correctness, or analyze its time complexity. Here our focus is on designing algorithms with invariants, not analyzing them.

As an example, consider the 2-sum problem. We are given an array A of sorted integers, and a target value K. We want to know if there exist entries i and j in A such that $A[i] + A[j] = K$.

The brute-force algorithm for the 2-sum problem consists of a pair of nested for loops. Its complexity is $O(n^2)$, where n is the length of A. A faster approach is to add each element of A to a hash H, and test for each i if $K - A[i]$ is present in H. While reducing time complexity to $O(n)$, this approach requires $O(n)$ additional storage for H.

We want to compute i and j such that $A[i] + A[j] = K$. Without loss of generality, we can take $i \le j$. We know that $0 \le i$, and $j \le n - 1$. A natural approach then is to initialize i to 0, and j to $n - 1$, and then update i and j preserving the following invariant:

   − No $i' < i$ can ever be paired with any j' such that $A[i'] + A[j'] = K$, and
   − No $j' > j$ can ever be paired with any i' such that $A[i'] + A[j'] = K$.

The invariant is certainly true at initialization, since there are no $i' < 0$ and $j' > n - 1$. To show how i and j can be updated while ensuring the invariant continues to hold, consider $A[i] + A[j]$. If $A[i] + A[j] = K$, we are done. Otherwise, consider the case $A[i] + A[j] < K$. We know from the invariant that for no $j' > j$ is there a solution in which the element with the larger index is j'. The element at i cannot be paired with any element at an index j' smaller than j—because A is sorted, $A[i] + A[j'] \le A[i] + A[j] < K$. Therefore, we can increment i, and preserve the invariant. Similarly, in the case $A[i] + A[j] > K$, we can decrement j and preserve the invariant.

We terminate when either $A[i] + A[j] = K$ (success) or $i > j$ (failure). At each step, we increment or decrement i or j. Since there are at most n steps, and each takes $O(1)$ time, the time complexity is $O(n)$. Correctness follows from the fact that the invariant never discards a value for i or j which could possibly be the index of an element which sums with another element to K.

Identifying the right invariant is an art. Usually, it is arrived at by studying concrete examples and then making an educated guess. Often the first invariant is too strong, i.e., it does not hold as the program executes, or too weak, i.e., it holds throughout the program execution but cannot be used to infer the result.

In some cases 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" which is propagated through the computation. The efficient frontier can be viewed as an invariant.

For example, suppose we need to implement a stack that supports the `max()` method, which is defined to return the largest value stored in the stack. We can associate for each entry in the stack the largest value stored at or below that entry.

This makes returning the largest value in the stack trivial. The invariant is that the value associated for each entry is the largest value stored at or below that entry. The invariant certainly continues to hold after a pop. To ensure the invariant holds after a push, we compare the value v being pushed with the largest value m stored in the stack prior to the push (which is the value associated with the entry currently at the top of the stack), and associate the entry being pushed with the larger of v and m.

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, summarized in Table 4.3, that are especially effective on common interview problems.

Table 4.3: Abstract analysis techniques.

Analysis principle	Key points
Concrete examples	Manually solve concrete instances of the problem and then build a general solution.
Case analysis	Split the input/execution into a number of cases and solve each case in isolation.
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.

CONCRETE EXAMPLES

Problems that seem difficult to solve in the abstract can become much more tractable when you examine concrete instances. Specifically, the following types of inputs can offer tremendous insight:

 – small inputs, such as an array or a BST containing 5–7 elements;
 – extreme/specialized inputs, e.g., binary values, nonoverlapping intervals, sorted arrays, connected graphs, etc.

Problems 5.5 on Page 44 and 16.1 on Page 76 are illustrative of small inputs, and Problem 5.6 on Page 44 is illustrative of extreme/specialized inputs.

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 4.3. 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 5.5 on Page 108 develops this analysis in more detail.

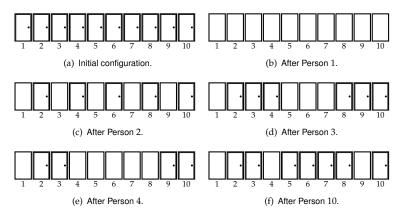|	
(a) Initial configuration.	(b) After Person 1.
(c) After Person 2.	(d) After Person 3.
(e) After Person 4.	(f) After Person 10.

Figure 4.3: Progressive updates to 10 doors.

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 mod 9 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 $\{\max S_1, \ldots, \max S_5\}$. 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.

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 $O(n)$, 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:

```
1  void rearrange(vector<int>* A) {
2    vector<int>& B = *A;
3    for (size_t i = 1; i < B.size(); ++i) {
4      if ((!(i & 1) && B[i - 1] < B[i]) || ((i & 1) && B[i - 1] > B[i])) {
5        swap(B[i - 1], B[i]);
6      }
7    }
8  }
```

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: 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.

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 79.

REDUCTION

Consider the problem of determining 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.

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.

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 USD → 1 × 0.8123 = 0.8123 EUR → 0.8123 × 1.2010 = 0.9755723 CHF → 0.9755723 × 80.39 = 78.426257197 JPY → 78.426257197 × 0.0128 = 1.00385609212 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. The solutions to the problems of painting a Boolean matrix (Problem 19.2 on Page 90) and string transformation (Problem 19.3 on Page 90) also illustrate modeling with graphs.

System design patterns

Sometimes, you will be asked how to go about creating a set of services or a larger system on top of an algorithm that you have designed. We summarize patterns that are useful for designing systems in Table 4.5.

Table 4.5: System design patterns.

Design principle	Key points
Decomposition	Split the functionality, architecture, and code into manageable, reusable components.
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.

Decomposition

Good decompositions are critical to successfully solving system-level design problems. Functionality, architecture, and code all benefit from decomposition.

For example, in our solution to designing a system for online advertising, we decompose the goals into categories based on the stake holders. We decompose the architecture itself into a front-end and a back-end. The front-end is divided into user management, web page design, reporting functionality, etc. The back-end is made up of middleware, storage, database, cron services, and algorithms for ranking ads.

Decomposing code is a hallmark of object-oriented programming. The subject of design patterns is concerned with finding good ways to achieve code-reuse. Broadly speaking, design patterns are grouped into creational, structural, and behavioral patterns. Many specific patterns are very natural—strategy objects, adapters, builders, etc., appear in a number of places in our codebase. Freeman *et al.*'s *"Head First Design Patterns"* is, in our opinion, the right place to study design patterns.

<small>PARALLELISM</small>

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
- each subproblem can be solved relatively independently, and
- 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.

The solution to Problem also illustrates the use of parallelism.

<small>CACHING</small>

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.

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 indicates an upper bound on running time.

As an example, searching an unsorted array of integers of length n, for a given integer, has an asymptotic complexity of $O(n)$ 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.

If the run time is asymptotically proportional to $f(n)$, the time complexity is written as $\Theta(f(n))$. The big-Omega notation, $\Omega(f(n))$, is used to denote an asymptotic lower bound of $f(n)$ on the time complexity of an algorithm. The big-Omega notation is illustrated by the $\Omega(n \log n)$ lower bound on any comparison-based array sorting algorithm. We follow the widespread custom of using the big-O notation where Θ or Ω would be more appropriate.

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 in a similar manner 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 $O(n)$ 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:

- It is very difficult to define meaningful distributions on the inputs.
- 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.
- Malicious users may exploit bad worst-case performance to create denial-of-service attacks.

Many authors, ourselves included, will refer to the time complexity of an algorithm as its complexity without the time qualification. The space complexity is

always qualified as such.

Intractability

In real-world settings you will often encounter problems that can be solved using efficient algorithms such as binary search and shortest paths. As we will see in the coming chapters, it is often difficult to identify such problems because the algorithmic core is obscured by details. Sometimes, 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.

Occasionally, 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 a certain class of formulas expressing 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 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 only the starting 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, including dynamic programming, which have exponential time complexity, may be acceptable, if the instances encountered are small, or if the specific parameter that the complexity is exponential in is small;
- search algorithms, such as backtracking, branch-and-bound, and hill-climbing, which prune much of the complexity of a brute-force search;
- approximation algorithms which return a solution that is provably close to optimum;
- 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 .
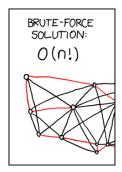
Figure 4.4: $P = NP$, by XKCD.

Part II

Problems

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 its instructions. 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 performed on it.

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. The width of a primitive-type variable is the number of bits of storage it takes in memory. For example, most implementations of C++ use 32 or 64 bits for an `int`. In Java an `int` is always 32 bits.

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 COMPUTE 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?

5.2   COMPUTE x/y 👁

Problem 5.2: Given two positive integers x and y, how would you compute x/y if the only operators you can use are addition, subtraction, and shifting?   *pg. 105*

5.3   CONVERT BASE

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 5.3: 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. 106*

5.4   GENERATE UNIFORM RANDOM NUMBERS

This problem is motivated by the following. Five friends have to select a designated driver using a single unbiased coin. The process should be fair to everyone.

Problem 5.4: 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?   *pg. 107*

5.5   THE OPEN DOORS 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 position of every i-th door starting from door i.

Problem 5.5: Which doors are open after the 500-th person has walked through?
          pg. 108

5.6   COMPUTE THE 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 5.6: Design an efficient algorithm for computing the GCD of two numbers without using multiplication, division or the modulus operators.   *pg. 108*

6

Aryays

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 $(i+1)$-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 THE DUTCH NATIONAL FLAG PROBLEM

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.

(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 of length n 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 $O(n)$ time complexity.   *pg. 109*

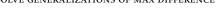## 6.2 COMPUTE THE MAX DIFFERENCE

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 6.2: 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. 110*

6.3 SOLVE GENERALIZATIONS OF MAX DIFFERENCE 😎

Problem 6.2, 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 6.3:   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 $\lfloor n/2 \rfloor$.

6.4   Sample offline data

Problem 6.4:   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.

6.5   Sample online data

This problem is motivated by the design of a packet sniffer that provides a uniform sample of packets for a network session.

Problem 6.5:   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.

Multidimensional arrays

Thus far we have focused our attention in this chapter on one-dimensional arrays. We now turn our attention to multidimensional arrays. A 2D array in an array whose entries are themselves arrays; the concept generalizes naturally to k dimensional arrays.

Multidimensional arrays arise in image processing, board games, graphs, modeling spatial phenomenon, etc. Often, but not always, the arrays that constitute the entries of a 2D array A have the same length, in which case we refer to A as being an $m \times n$ rectangular array (or sometimes just an $m \times n$ array), where m is the number of entries in A, and n the number of entries in $A[0]$. The elements within a 2D array A are often referred to by their *row* and *column* indices i and j, and written as $A[i][j]$ or $A[i, j]$.

7

Strings

String pattern matching is an important problem that occurs in many areas of science and information processing. In computing, it occurs naturally as part of data processing, text editing, term rewriting, lexical analysis, and information retrieval.

— *"Algorithms For Finding Patterns in Strings,"*
A. V. AHO, 1990

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. We now present problems on strings which can be solved using elementary techniques. Advanced string processing algorithms often use hash tables (Chapter 13) and dynamic programming (Page 79).

7.1 INTERCONVERT STRINGS AND INTEGERS

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 7.1: Implement string/integer inter-conversion functions. Use the following function signatures: String intToString(int x) and int stringToInt(String s). *pg. 115*

7.2 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 7.2: Implement a function for reversing the words in a string *s*. Your function should use $O(1)$ space.   *pg. 116*

7.3   Compute all mnemonics for a phone number

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 7.1. 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 7.1: Phone keypad.

Problem 7.3: Write a function which takes as input a phone number, specified as a string of digits, return all possible character sequences that correspond to the phone number. The cell phone keypad is specified by a mapping *M* that takes a digit and returns the corresponding set of characters. The character sequences do not have to be legal words or phrases.   *pg. 117*

8

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 $\wedge$ 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 8.1. 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 8.2. Since lists can be defined recursively, recursion is a natural candidate for list manipulation.
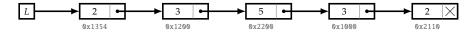
Figure 8.1: Example of a singly linked list. The number in hex below a node indicates the memory address of that node.
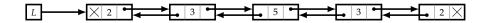
Figure 8.2: Example of a doubly linked list.

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:

```cpp
1  template <typename T>
```

```
2 struct ListNode {
3   T data;
4   shared_ptr<ListNode<T>> next;
5 };
```

8.1   MERGE TWO SORTED LISTS

Let L and F be singly linked lists in which each node holds a number. Assume the numbers in L and F appear in ascending order within the lists. The *merge* of L and F is a list consisting of the nodes of L and F in which numbers appear in ascending order. The merge function is shown in Figure 8.3.
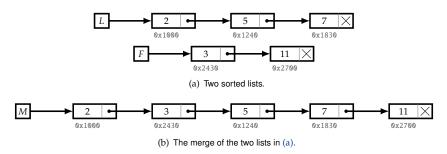

(a) Two sorted lists.

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

Figure 8.3: Merging sorted lists.

Problem 8.1: 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 8.3. The only field you can change in a node is `next`.   *pg. 117*

8.2   REVERSE 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.

Problem 8.2: Give a linear time nonrecursive function that reverses a singly linked list. The function should use no more than constant storage beyond that needed for the list itself.   *pg. 118*

8.3   TEST 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 8.3: 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. 119*

8.4 COPY 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 8.4 is a postings list with four nodes.
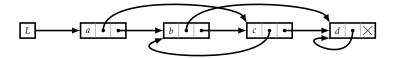
Figure 8.4: A postings list.

Problem 8.4: 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. 121*

9

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 9.1. 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).
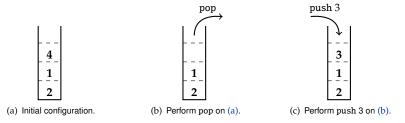
Figure 9.1: Operations on a stack.

9.1   IMPLEMENT A STACK WITH MAX API

Problem 9.1: Design a stack that supports a max operation, which returns the maxi-
mum 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. If the stack contains n
elements, you can use $O(n)$ space, in addition to what is required for the elements
themselves.   *pg. 123*

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 9.3 for details.

(a) Initial configuration.   (b) Queue (a) after dequeue.   (c) Queue (b) after enqueue(4).
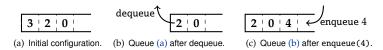
Figure 9.2: Examples of enqueue and dequeue.

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.

9.2 PRINT A BINARY TREE IN ORDER OF INCREASING DEPTH

Binary trees are formally defined in Chapter 10. In particular, each node in a binary tree has a depth, which is its distance from the root.

Problem 9.2: Given the root node r of a binary tree, print all the keys at r and its descendants. The keys should be printed in the order of the corresponding nodes' depths. Specifically, all keys corresponding to nodes of depth d should appear in a single line, and the next line should correspond to keys corresponding to nodes of depth $d + 1$. You cannot use recursion. You may use a single queue, and constant additional storage. For example, you should print
```
314
6 6
271 561 2 271
28 0 3 1 28
17 401 257
641
```
for the binary tree in Figure 10.1 on Page 57.   *pg. 126*

9.3 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 9.3: 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.

10

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 10.1 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.
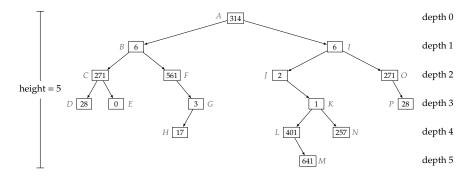
Figure 10.1: Example of a binary tree. The node depths range from 0 to 5. Node M has the highest depth (5) of any node in the tree, implying the height of the tree is 5.

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

```
1  template <typename T>
2  struct BinaryTreeNode {
3    T data;
4    unique_ptr<BinaryTreeNode<T>> left, right;
5  };
```

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. Usually, but not universally, the node object definition includes 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. A *level* of a tree is all nodes at the same depth. See Figure 10.1 on the preceding page for an example of the depth and height concepts.

As concrete examples of these concepts, consider the binary tree in Figure 10.1 on the previous 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, 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 nonleaf 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 *traversing* all the nodes in the tree. (Traversing is also sometimes called *walking*.) Here are some ways in which this visit can be done.

- Traverse the left subtree, visit the root, then traverse the right subtree (an *inorder* traversal). An inorder traversal of the binary tree in Figure 10.1 on the preceding page visits the nodes in the following order: $\langle D, C, E, B, F, H, G, A, J, L, M, K, N, I, O, P \rangle$.
- Visit the root, traverse the left subtree, then traverse the right subtree (a *preorder* traversal). A preorder traversal of the binary tree in Figure 10.1 on the previous page visits the nodes in the following order: $\langle A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P \rangle$.

– Traverse the left subtree, traverse the right subtree, and then visit the root (a *postorder* traversal). A postorder traversal of the binary tree in Figure 10.1 on Page 57 visits the nodes in the following order: $\langle D, E, C, H, G, F, B, M, L, N, K, J, P, O, I, A \rangle$.

Let T be a binary tree on n nodes, with height h. Implemented recursively, these traversals 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 traversals can be done with $O(1)$ additional space complexity.

Remarkably, an inorder traversal 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.

```
1  void inorder_traversal(const unique_ptr<BinaryTreeNode<int>>& root) {
2    auto* n = root.get();
3    while (n) {
4      if (n->left.get()) {
5        // Finds the predecessor of n.
6        auto* pre = n->left.get();
7        while (pre->right.get() && pre->right.get() != n) {
8          pre = pre->right.get();
9        }
10
11       // Processes the successor link.
12       if (pre->right.get()) {  // pre->right.get() == n.
13         // Reverts the successor link if predecessor's successor is n.
14         pre->right.release();
15         cout << n->data << endl;
16         n = n->right.get();
17       } else {  // if predecessor's successor is not n.
18         pre->right.reset(n);
19         n = n->left.get();
20       }
21     } else {
22       cout << n->data << endl;
23       n = n->right.get();
24     }
25   }
26 }
```

The term tree is overloaded, which can lead to confusion; see Page 89 for an overview of the common variants.

10.1 TEST IF A BINARY TREE IS BALANCED

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. A perfect binary tree is balanced, as is a complete binary tree. A balanced binary tree does not have to be perfect or complete—see Figure 10.2 on the following page for an example.
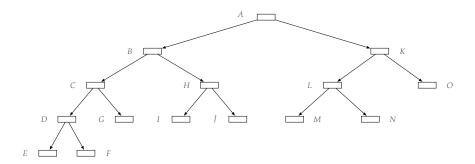
Figure 10.2: A balanced binary tree of height 4.

Problem 10.1: 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. 128***

10.2   COMPUTE THE LCA 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 10.1 on Page 57 is *K*.

Problem 10.2: 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. 129***

10.3   IMPLEMENT AN INORDER TRAVERSAL WITH $O(1)$ SPACE

The direct implementation of an inorder traversal 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 10.3: 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. 130***

10.4   COMPUTE THE SUCCESSOR

The successor of a node *n* in a binary tree is the node *s* that appears immediately after *n* in an inorder traversal. For example, in Figure 10.1 on Page 57, the successor of Node *G* is Node *A*, and the successor of Node *A* is Node *J*.

Problem 10.4: Design an algorithm that takes a node n in a binary tree, and returns its successor. Assume that each node has a `parent` field; the `parent` field of root is `null`. *pg. 131*

11

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.

11.1 MERGE SORTED FILES

You are given 500 files, each containing stock trade information for an S&P 500 company. Each trade is encoded by a line 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 in increasing order of time. The remaining values are the stock symbol, number of shares, and price. You are to create a single file containing all the trades from the 500 files, sorted in order of increasing 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 11.1: Design an algorithm that takes a set of files containing stock trades sorted by increasing trade times, and writes a single file containing the trades appearing in the individual files sorted in the same order. The algorithm should use very little RAM, ideally of the order of a few kilobytes. *pg. 131*

11.2 COMPUTE THE *k* 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 11.2: How would you compute the *k* stars which are closest to the Earth? You have only a few megabytes of RAM.

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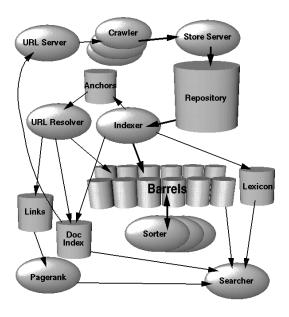
You want to compute the running median of a sequence of numbers. The sequence is presented to you in a streaming fashion—you cannot back up to read an earlier value, and you need to output the median after reading in each new element.

Problem 11.3: 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.

12

Searching

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

Search algorithms can be classified in a number of ways. Is the underlying collection static or dynamic, i.e., inserts and deletes are interleaved with searching? Is worth spending the computational cost to preprocess the data so as to speed up subsequent queries? Are there statistical properties of the data that can be exploited? Should we operate directly on the data or transform it?

In this chapter, our focus is on static data stored in sorted order in an array. Data structures appropriate for dynamic updates are the subject of Chapters 11, 13, and 15.

The first collection of problems in this chapter are related to binary search. The second collection pertains to general search.

Binary search

Binary search is at the heart of more interview questions than any other single algorithm. 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 $O(n)$ time complexity.

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(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:

```
1  #define N 3000000000
2  char A[N];
```

```
3  char* B = (A + 1500000000);
4  int L = -1499000000;
5  int U = 1499000000;
6  // On a 32-bit machine (U - L) = -1296967296 because the actual value,
7  // 2998000000 is larger than 2^31 - 1. Consequently, the bsearch function
8  // called below sets m to -2147483648 instead of 0, which leads to an
9  // out-of-bounds access, since the most negative index that can be applied
10 // to B is -1500000000.
11 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.

12.1   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
$A[0]$	$A[1]$	$A[2]$	$A[3]$	$A[4]$	$A[5]$	$A[6]$	$A[7]$	$A[8]$	$A[9]$
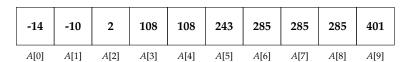
Figure 12.1: A sorted array with repeated elements.

Problem 12.1: 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 12.1 your algorithm should return 3 if $k = 108$; if $k = 285$, your algorithm should return 6.   *pg. 135*

12.2   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 12.2 on the next page.

Problem 12.2: 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 12.2 on the facing page, your algorithm should return 4. *pg. 135*
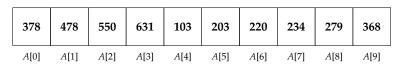
378	478	550	631	103	203	220	234	279	368
A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]

Figure 12.2: A cyclically sorted array.

12.3 SEARCH 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 12.3: 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. 136*

Generalized search

Now we consider a number of search problems that do not use the binary search principle. For example, they focus on tradeoffs between RAM and computation time, avoid wasted comparisons when searching for the minimum and maximum simultaneously, use randomization to perform elimination efficiently, use bit-level manipulations to identify missing elements, etc.

12.4 FIND THE MISSING IP ADDRESS

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

Problem 12.4: 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. 138*

13

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 ($O(n + m)$ 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 15), 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.

A hash function has one hard requirement—equal keys should have equal hash codes. 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, or by including profiling data.

A softer requirement is that the hash function should "spread" keys, i.e., the hash codes for a subset of objects should be uniformly distributed across the underlying array. In addition, a hash function should be efficient to compute.

Now we illustrate the steps in designing a hash function suitable for strings. First, the hash function should examine all the characters in the string. (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. The following function has these properties:

```
1  int string_hash(const string& str, int modulus) {
2    const int kMult = 997;
3    int val = 0;
4    for (const char& c : str) {
5      val = (val * kMult + c) % modulus;
6    }
7    return val;
8  }
```

A hash table is a good data structure to represent a dictionary, i.e., a set of strings. In some applications, a trie, which is an tree data structure that is used to store a dynamic set of strings. Unlike a BST, nodes in the tree do not store a key. Instead, the node's position in the tree defines the key which it is associated with.

13.1 PARTITION INTO 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 13.1: 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. 138*

13.2 TEST IF AN ANONYMOUS LETTER IS CONSTRUCTIBLE

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

Problem 13.2: 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. 139*

13.3 FIND THE LINE THROUGH THE MOST POINTS 👁

Problem 13.3: 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. 140*

14

Sorting

PROBLEM 14 *(Meshing). Two monotone sequences S, T, of lengths n, m, respectively, are stored in two systems of n(p + 1), m(p + 1) consecutive memory locations, respectively: s, s + 1, ..., s + n(p + 1) − 1 and t, t + 1, ..., t + m(p + 1) − 1. ... It is desired to find a monotone permutation R of the sum [S, T], and place it at the locations r, r + 1, ..., r + (n + m)(p + 1) − 1.*

— *"Planning And Coding Of Problems For An Electronic Computing Instrument,"*
H. H. GOLDSTINE AND J. VON NEUMANN, 1948

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 $O(n^2)$ 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.)

A well-implemented quicksort is usually the best choice for sorting. We briefly outline alternatives that are better in specific circumstances.

For short arrays, e.g., 10 or fewer elements, insertion sort is easier to code and faster than asymptotically superior sorting algorithms. 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. If there are a small number of distinct keys, e.g., integers in the range [0..255], counting sort, which records for each element, the number of elements less than it, works well. 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. If there are many duplicate keys 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 traversal of the BST.

Most 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.

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 11. 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 Problem 11.1 on Page 62.

14.1 COMPUTE THE INTERSECTION OF 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 14.1: 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. 142

14.2 RENDER 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 nonoverlapping 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 14.1 on the facing page 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.

Problem 14.2: Given a set of n events, how would you determine the maximum number of events that take place concurrently?
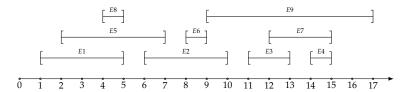*pg. 143*

Figure 14.1: 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.

14.3 ADD A CLOSED INTERVAL

Consider a set U consisting of disjoint closed intervals. This problem is concerned with adding an interval I to U and computing a new set of disjoint and sorted intervals containing exactly the numbers in $U \cup \{I\}$. For example, if $U = \langle[0, 2], [3, 6], [7, 7], [9, 12]\rangle$, and $I = [1, 8]$ is added, the result should be $\langle[0, 8], [9, 12]\rangle$.

Problem 14.3: Write a function which takes as input an array A of disjoint closed intervals with integer endpoints, sorted by increasing order of left endpoint, and an interval I, and returns the union of I with the intervals in A, expressed as a union of disjoint intervals. *pg. 144*

15

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 10 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 15.1 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 $O(n)$, if insertions and deletions are naïvely implemented. However, there are implementations of insert and delete which guarantee the tree has height $O(\log n)$. 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:

```cpp
template <typename T>
struct BSTNode {
  T data;
  unique_ptr<BSTNode<T>> left, right;
};
```

15.1 TEST IF A BINARY TREE SATISFIES THE BST PROPERTY

Problem 15.1: 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. 145*
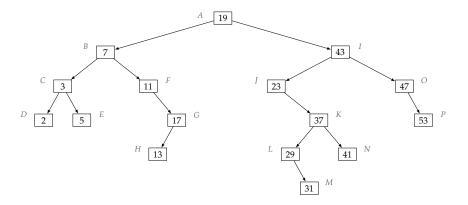
Figure 15.1: An example BST.

15.2 Find the first key larger than k in a BST

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 15.2: 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 traversal. If k is absent or no key larger than k is present, return null. For example, when applied to the BST in Figure 15.1 you should return 29 if $k = 23$; if $k = 32$, you should return null.

15.3 Build 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 15.3: How would you build a BST of minimum possible height from a sorted array A?

CHAPTER

16

Recursion

The power of recursion evidently lies in the possibility of defining an infinite set of objects by a finite statement. In the same manner, an infinite number of computations can be described by a finite recursive program, even if this program contains no explicit repetitions.

— *"Algorithms + Data Structures = Programs,"*
N. E. WIRTH, 1976

Recursion is a method where the solution to a problem depends partially on solutions to smaller instances of related problems. Two key ingredients to a successful use of recursion are identifying the base cases, which are to be solved directly, and ensuring progress, that is the recursion converges to the solution.

Recursion can be applied to many types of problems. As described on Page 28, recursion is especially suitable when the input is expressed using recursive rules.

In this chapter we study recursion in a general form, as well as its application to enumeration and divide-and-conquer. Recursion is suitable for solving computationally intractable problems (Page on Page 40). Both backtracking and branch-and-bound are naturally formulated using recursion; Problem 16.3 illustrates them.

Chapter 17 describes dynamic programming, which conceptually is based on recursion augmented with a cache to avoid solving the same problem multiple times.

16.1 THE TOWERS OF HANOI PROBLEM

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 16.1 on the next page. 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.

Problem 16.1: 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. 149*

16.2 ENUMERATE 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 16.2 on the facing page.

(a) Initial configuration. (b) Desired configuration.
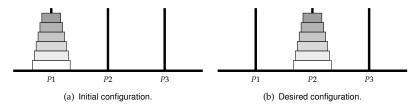
Figure 16.1: Towers of Hanoi for $n = 6$.
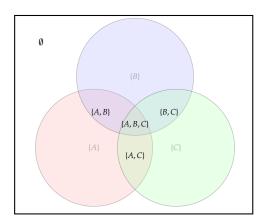
Figure 16.2: The power set of $\{A, B, C\}$ is $\{\emptyset, \{A\}, \{B\}, \{C\}, \{A, B\}, \{B, C\}, \{A, C\}, \{A, B, C\}\}$.

Problem 16.2: Implement a method that takes as input a set S of n distinct elements, and prints the power set of S. Print the subsets one per line, with elements separated by commas. *pg. 151*

16.3 IMPLEMENT A SUDOKU SOLVER

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 16.3: 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. 153*

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 11.1 on Page 62 and 6.1 on Page 46 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. 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 16.3 shows how this algorithm can lead to suboptimal results.

(a) A weighted undirected graph.   (b) An MST built from the MSTs on   (c) An optimum MST.
                                          $\{a, b, c\}$ and $\{d, e.f\}$.
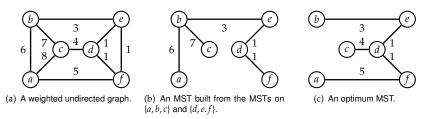
Figure 16.3: Divide-and-conquer applied to the MST problem is suboptimum.

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.

Divide-and-conquer is not synonymous with recursion. First of all, in divide-and-conquer, the problem is divided into two or more independent smaller problems that are of the same type as the original problem. Recursion is more general—there may be a single subproblem, e.g., binary search, the subproblems may not be independent, e.g., dynamic programming, and they may not be of the same type as the original, e.g., regular expression matching. In addition, sometimes in order to improve runtime, and occasionally to reduce space complexity, a divide-and-conquer algorithm is implemented using iteration instead of recursion. The iterative implementation mimics the recursion, perhaps using a stack. Invariably, the iterative version is more challenging to implement correctly.

17

Dynamic Programming

> *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

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 17.1 on the following page 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 17.1: 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 l for any subarray ending at the last entry in L, and the maximum subarray sum r for any subarray starting at 0 for 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 an $O(n \log n)$.

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 $S[j] - \min_{i \leq j} S[i]$. 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 $O(n)$ time and $O(1)$ space solution. The code below returns a pair of indices (i, j) such that $A[i : j-1]$ is a maximum subarray. It is legal for all array entries to be negative, or the array to be empty. The algorithm handles these input cases correctly. Specifically, it returns equal indices, which denote an empty subarray.

```
1  pair<int, int> find_maximum_subarray(const vector<int>& A) {
2    // A[range.first : range.second - 1] will be the maximum subarray.
3    pair<int, int> range(0, 0);
4    int min_idx = -1, min_sum = 0, sum = 0, max_sum = 0;
5    for (int i = 0; i < A.size(); ++i) {
6      sum += A[i];
7      if (sum < min_sum) {
8        min_sum = sum, min_idx = i;
9      }
10     if (sum - min_sum > max_sum) {
11       max_sum = sum - min_sum, range = {min_idx + 1, i + 1};
12     }
13   }
14   return range;
15 }
```

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. However, in most cases, these two subproblems are not sufficient to solve the original problem.

17.1 COUNT THE NUMBER OF 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 17.1: 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. 155*

17.2 COUNT THE NUMBER OF WAYS TO TRAVERSE A 2D ARRAY

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 17.2.
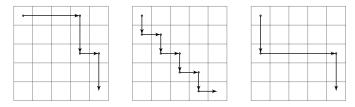
Figure 17.2: Paths through a 2D array.

Problem 17.2: 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. 156*

17.3　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 17.3. 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.

| A, \$65, 20 oz. | B, \$35, 8 oz. | C, \$245, 60 oz. | D, \$195, 55 oz. | E, \$65, 40 oz. | F, \$150, 70 oz. | G, \$275, 85 oz. | H, \$155, 25 oz. |

| I, \$120, 30 oz. | J, \$320, 65 oz. | K, \$75, 75 oz. | L, \$40, 10 oz. | M, \$200, 95 oz. | N, \$100, 50 oz. | O, \$220, 40 oz. | P, \$99, 10 oz. |
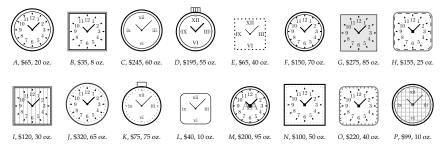
Figure 17.3: A clock store.

Problem 17.3: 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. 158**

17.4　The bedbathandbeyond.com problem

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 17.4: Given a dictionary, i.e., a set of strings, 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. 158**

17.5　Find the 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 17.4 on the next page 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
$A[0]$	$A[1]$	$A[2]$	$A[3]$	$A[4]$	$A[5]$	$A[6]$	$A[7]$	$A[8]$	$A[9]$
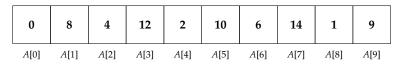
Figure 17.4: An array whose longest nondecreasing subsequences are of length 4.

Problem 17.5: 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 $A[i_j] \le A[i_{j+1}]$ for any $j \in [0, k-2]$.   *pg. 159*

18

Greedy Algorithms and Invariants

The intended function of a program, or part of a program, can be specified my making general assertions about the values which the relevant variables will take after *execution of the program.*

— *"An Axiomatic Basis for Computer Programming,"*
C. A. R. HOARE, 1969

Greedy algorithms

As described on Page 31, a greedy algorithm is an algorithm that computes a solution in steps; at each step the algorithm makes a decision that is locally optimum, and it never changes that decision.

The example on Page 31 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 on Page 38, 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 17.3 on Page 82.

18.1 IMPLEMENT 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 18.1 shows the large variation in the frequencies of letters of the English alphabet.

Table 18.1: 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 18.1: Given a set of symbols with corresponding frequencies, find a code book that has the smallest average code length. *pg. 162*

Invariants

An invariant is a condition that is true during execution of a program. Invariants can be used to design algorithms as well as reason about their correctness.

The reduce and conquer algorithms seen previously, e.g., binary search, maintain the invariant that the space of candidate solutions contains all possible solutions as the algorithms execute.

Sorting algorithms nicely illustrates algorithm design using invariants. For example, intuitively, selection sort is based on finding the smallest element, the next smallest element, etc. and moving them to their right place. More precisely, we work with successively larger subarrays beginning at index 0, and preserve the invariant that these subarrays are sorted and their elements are less than or equal to the remaining elements.

As another example, suppose we want to find two elements in a sorted array A summing to a specified K. Let n denote the length of A. We start by considering $s_{0,n-1} = A[0] + A[n-1]$. If $s_{0,n-1} = K$, we are done. If $s_{0,n-1} < K$, then we can restrict our attention to solving the problem on the subarray $A[1 : n-1]$, since $A[0]$ can never be one of the two elements. Similarly, if $s_{0,n-1} > K$, we restrict the search to $A[0 : n-2]$. The invariant is that if two elements which sum to K exist, they must lie within the subarray currently under consideration.

18.2 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 18.2: Design an algorithm that takes as input an array A and a number t, and determines if A 3-creates t.   *pg. 164*

18.3 COMPUTE THE 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 18.1, 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).
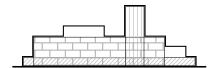
Figure 18.1: Buildings, their skyline, and the largest contained rectangle.

Problem 18.3: 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. 165*

19

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 19.1.

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 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 19.1.
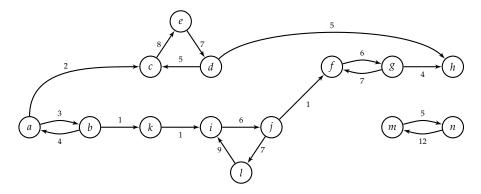
Figure 19.1: A directed graph with weights on edges.

A *directed acyclic graph* (DAG) is a directed graph in which there are no *cycles*, i.e., paths of the form $\langle v_0, v_1, \ldots, v_{n-1}, v_0 \rangle$, $n \geq 1$. See Figure 19.2 on the following page for an example of a directed acyclic graph. Vertices in a DAG which have no incoming

edges are referred to as *sources*; vertices which have no outgoing edges are referred to as *sinks*. A *topological ordering* of the vertices in a DAG is an ordering of the vertices in which each edge is from a vertex earlier in the ordering to a vertex later in the ordering.
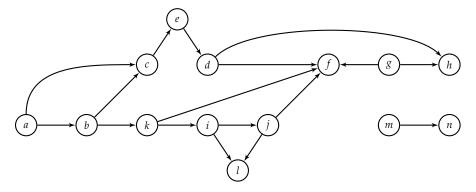
Figure 19.2: A directed acyclic graph. Vertices a, g, m are sources and vertices l, f, h, n are sinks. The ordering $\langle a, b, c, e, d, g, h, k, i, j, f, l, m, n \rangle$ is a topological ordering of the vertices.

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 19.3.
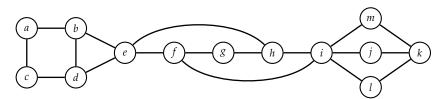
Figure 19.3: An undirected graph.

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 10, 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 19.4 is a tree. Note that its edge set is a subset of the edge set of the undirected graph in Figure 19.3 on the preceding page. 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.
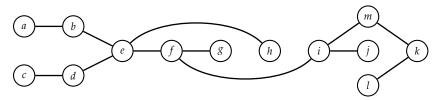
Figure 19.4: A tree.

Graph search

Computing vertices which are reachable from other vertices is a fundamental operation which can be performed in one of two idiomatic ways, namely depth-first search (DFS) and breadth-first search (BFS). Both have linear time complexity—$O(|V| + |E|)$ to be precise. In a worst case there is a path from the initial vertex covering all vertices without any repeats, and the DFS edges selected correspond to this path, so the space complexity of DFS is $O(|V|)$ (this space is implicitly allocated on the function call stack). The space complexity of BFS is also $O(|V|)$, since in a worst case there is an edge from the initial vertex to all remaining vertices, implying that they will all be in the BFS queue simultaneously at some point.

DFS and BFS 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.

19.1 SEARCH 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 19.5.

| (a) A maze. | (b) A path from entrance to exit. | (c) A shortest path from entrance to exit. |
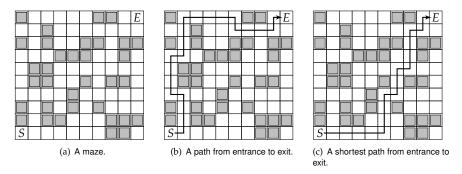
Figure 19.5: An instance of the maze search problem, with two solutions, where S and E denote the entrance and exit, respectively.

Problem 19.1: 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. 167*

19.2 PAINT A BOOLEAN MATRIX

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.

Problem 19.2: Implement a routine that takes a $n \times m$ Boolean array A together with an entry (x, y) and flips the color of the region associated with (x, y). See Figure 19.6 on the next page for an example of flipping.   *pg. 169*

19.3 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*.
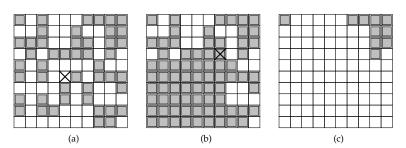
Figure 19.6: The color of all squares associated with the first square marked with a × in (a) have been recolored to yield the coloring in (b). The same process yields the coloring in (c).

Problem 19.3: 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. 170*

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 path*—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 problems.

19.4 Compute a 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 19.4: 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. 171*

20

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 while other threads are, for example, busy doing network communication and passing results to the UI thread, resulting in increased responsiveness), Java virtual machines (a separate thread handles garbage collection which would otherwise lead to blocking, while another thread is busy running the user code), 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 20.3),
- deadlock (Thread *A* acquires Lock *L*1 and Thread *B* acquires Lock *L*2, following which *A* tries to acquire *L*2 and *B* tries to acquire *L*1), 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.

20.1 IMPLEMENT SYNCHRONIZATION FOR TWO INTERLEAVING THREADS

Thread *T*1 prints odd numbers from 1 to 100; Thread *T*2 prints even numbers from 1 to 100.

Problem 20.1: Write Java code in which the two threads, running concurrently, print the numbers from 1 to 100 in order.   *pg. 172*

20.2 IMPLEMENT A TIMER CLASS

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 20.2: 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. 173*

20.3 THE READERS-WRITERS PROBLEM

Consider an object *s* which is read from and written to by many threads. 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 *R*1 has locked *s* and another reader *R*2 wants to access *s*. Reader *R*2 does not have to wait until *R*1 is done reading; instead, *R*2 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 20.3: Implement a synchronization mechanism for the first readers-writers problem. *pg. 173*

21

Design Problems

> *We have described a simple but very powerful and flexible pro-*
> *tocol 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. 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.

21.1 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.) Find X satisfying $X = \epsilon[\mathbf{1}] + (1 - \epsilon)A^T X$. Here ϵ is a constant, e.g., $\frac{1}{7}$, and $[\mathbf{1}]$ 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: $X_k = \epsilon[\mathbf{1}] + (1 - \epsilon)A^T X_{k-1}$.

Problem 21.1: Design a system that can compute the ranks of ten billion web pages in a reasonable amount of time. *pg. 175*

21.2 IMPLEMENT MILEAGE RUN

Airlines often give customers who fly frequently with them a "status". This status allows them early boarding, more baggage, upgrades to executive class, etc. Typically, status is a function of miles flown in the past twelve months. People who travel frequently by air sometimes want to take a round trip flight simply to maintain their status. The destination is immaterial—the goal is to minimize the cost-per-mile (cpm), i.e., the ratio of dollars spent to miles flown.

Problem 21.2: Design a system that will help its users find mileage runs. *pg. 175*

Part III

Hints

Hints

Use a hint after you have made a serious attempt at the problem. Ideally, the hint should give you the flash of insight needed to complete your solution.

Usually, you will receive hints only after you have shown an understanding of the problem, and have made serious attempts to solve it. See Chapter 2 for strategies on conducting yourself at the interview.

5.1: Use a lookup table, but don't use 2^{64} entries!

5.2: Relate x/y to $(x - y)/y$.

5.3: What base can you easily convert to and from?

5.4: How would you mimic a three-sided coin with a two-sided coin?

5.5: Solve the same problem with 2, 5, 10, and 20 doors.

5.6: Use case analysis: both even; both odd; one even and one odd.

6.1: Think about the partition step in quicksort.

6.2: Identifying the minimum and maximum heights is not enough since the minimum height may appear after the maximum height. Focus on valid height differences.

6.3: What do you need to know about $A[0 : i - 1]$ when processing $A[i]$?

6.4: Use a routine that yields k-sized subsets to create a routine for $k + 1$-sized subsets.

6.5: Suppose you have a procedure which selects k packets from the first $n \geq k$ packets as specified. How would you deal with the $(n + 1)$-th packet?

7.1: Build the result one digit at a time.

7.2: It's difficult to solve this with one pass.

7.3: Use recursion.

8.1: Two sorted arrays can be merged using two indices. For lists, take care when one pointer variable reaches the end.

8.2: Use a pair of pointers.

8.3: Consider using two pointers, one fast and one slow.

8.4: Copy the jump field and then copy the next field.

9.1: Use additional storage to track the maximum value.

9.2: First think about solving this problem with a pair of queues.

9.3: Track the head and tail. How can you differentiate a full queue from an empty one?

10.1: Think of a classic binary tree algorithm that runs in $O(h)$ additional space.

10.2: When is the root the LCA?

10.3: How can you tell whether a node is a left child or right child of its parent?

10.4: Study n's right subtree. What if n does not have a right subtree?

11.1: Which portion of each file is significant as the algorithm executes?

11.2: Suppose you know the k closest stars in the first n stars. If the $(n + 1)$-th star is to be added to the set of k closest stars, which element in that set should be evicted?

11.3: Can you avoid tracking all elements?

12.1: Don't stop after you reach the first k. Think about the case where every entry equals k.

12.2: Use the decrease and conquer principle.

12.3: The first k elements of A together with the first k elements of B are initial candidates. Iteratively eliminates a constant fraction of the candidates.

12.4: Can you be sure there is an address which is not in the file?

13.1: Map strings to strings so that strings which are anagrams map to the same string.

13.2: Count.

13.3: A line can be uniquely represented by two numbers.

14.1: Solve the problem if n and m differ by orders of magnitude. What if $n \approx m$?

14.2: Focus on endpoints.

14.3: What is the union of two closed intervals?

15.1: Is it correct to check for each node that its key is greater than or equal to the key at its left child and less than or equal to the key at its right child?

15.2: Perform binary search, keeping some additional state.

15.3: Which element should be the root?

16.1: If you know how to transfer the top $n - 1$ rings, how does that help move the n-th ring?

16.2: There are 2^n subsets for a given set S of size n. There are 2^k k-bit words.

16.3: Apply the constraints to speed up a brute-force algorithm.

17.1: Count the number of combinations in which there are 0 w_0 plays, then 1 w_0 plays, etc.

17.2: If $i > 0$ and $j > 0$, you can get to (i, j) from $(i - 1, j)$ or $(j - 1, i)$.

17.3: The "obvious" recurrence is not the right one.

17.4: Solve the generalized problem, i.e., determine for each prefix of s whether it is the concatenation of dictionary words.

17.5: Express the longest nondecreasing subsequence ending at $A[i]$ in terms of the longest nondecreasing subsequence in $A[0 : i - 1]$.

18.1: Reduce the problem from n symbols to one on $n - 1$ symbols.

18.2: How would you check if $A[i]$ is part of a triple that 3-creates t?

18.3: How would you efficiently find the largest rectangle which includes the i-th building, and has height $A[i]$?

19.1: Model the maze as a graph.

19.2: Solve this conceptually, then think about implementation optimizations.

19.3: Treat strings as vertices in an undirected graph, with an edge between u and v iff the corresponding strings differ in one character.

19.4: What property does a minimal set of infeasible tasks have?

20.1: The two threads need to notify each other when they are done.

20.2: There are two aspects—data structure design and concurrency.

20.3: Track the number of readers.

21.1: This must be performed on an ensemble of machines. The right data structures will simplify the computation.

21.2: Partition the implied features into independent tasks.

Part IV

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.
– An initializer list uses the `{}` notation to avoid having to make explicit calls to constructors when building list-like objects.

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 13.3 on Page 140 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 `unique_ptr` is a smart pointer that retains sole ownership of an object through a pointer and destroys that object when the `unique_ptr` goes out of scope.
- A `shared_ptr` is a smart pointer with a reference count which the runtime system uses to implement automatic garbage collection.

Problem 5.1, *pg. 43*: *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. Let n denote the width of an integer and k the number of bits set to 1 set in a particular integer. (For example, for a 64-bit integer, $n = 64$. For the specific integer $(1011)_2$, $k = 3$.)

The brute-force algorithm consists of iteratively testing the value of each bit.

```
1  short parity1(unsigned long x) {
2    short result = 0;
3    while (x) {
4      result ^= (x & 1);
5      x >>= 1;
6    }
7    return result;
8  }
```

The time complexity of the algorithm above is $O(n)$.

There is a neat trick that erases the lowest set bit of a number in a single operation which can be used to improve performance in the best and average cases.

```
1  short parity2(unsigned long x) {
2    short result = 0;
3    while (x) {
4      result ^= 1;
5      x &= (x - 1);   // erases the lowest set bit of x.
6    }
7    return result;
8  }
```

The time complexity of the algorithm above is $O(k)$.

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-based lookup table. The optimum size for the lookup table is a function of how much RAM is available, and how big the cache is. In the implementation below we store the parity of i, a 16-bit integer in `precomputed_parity[i]`. Each `precomputed_parity[i]` can be statically initialized. Alternately, we can use lazy initialization, with a separate flag bit used to indicate whether a particular `precomputed_parity[i]` value is valid. The following implementation of parity uses this approach. The time complexity is a function of the size of the keys used to index `precomputed_parity`. For a sufficiently large array,

it can be $O(1)$. (This does not include the time to initialize `precomputed_parity`.) The time complexity for general n is $O(n)$.

```
1  short parity3(unsigned long x) {
2    return precomputed_parity[x >> 48] ^
3           precomputed_parity[(x >> 32) & 0b1111111111111111] ^
4           precomputed_parity[(x >> 16) & 0b1111111111111111] ^
5           precomputed_parity[x & 0b1111111111111111];
6  }
```

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.)

Another implementation with a smaller lookup table is shown below. We make use of the property that parity is commutative. For example, the parity of $\langle b_{63}, b_{62}, \ldots, b_3, b_2, b_1, b_0 \rangle$ equals the parity of $\langle b_{63} \oplus b_{31}, b_{62} \oplus b_{30}, \ldots, b_{32} \oplus b_0 \rangle$; the latter 32 bit value can be computed with one shift and one XOR instruction. This leads to the algorithm below. The final step entails a lookup into a lookup table indexed by a 4 bit quantity—we could instead have performed two more shift and XOR steps.

```
1  short parity4(unsigned long x) {
2    x ^= x >> 32;
3    x ^= x >> 16;
4    x ^= x >> 8;
5    x ^= x >> 4;
6    x &= 0xf; // only want the last 4 bits of x.
7    // Return the LSB, which is the parity.
8    return four_bit_parity_lookup(x) & 1;
9  }
10
11 // The LSB of kFourBitParityLookupTable is the parity of 0,
12 // next bit is parity of 1, followed by the parity of 2, etc.
13
14 const int kFourBitParityLookupTable = 0x6996; // = 0b0110100110010110.
15
16 short four_bit_parity_lookup(int x) {
17   return kFourBitParityLookupTable >> x;
18 }
```

Problem 5.2, pg. 44: *Given two positive integers x and y, how would you compute x/y if the only operators you can use are addition, subtraction, and shifting?*

Solution 5.2: 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 $2^k + \frac{(x-2^k y)}{y}$. Note that $2^k y$ can be computed by left-shifting y by k.

Let n be the number of bits needed to represent x. Assume $x \geq y$. Each iteration reduces the dividend in the recursive call by at least half, so there are $O(n)$ recursive calls. If the largest k such that $2^k y \leq x$ is computed by iterating through k, each call has time complexity $O(n)$. This leads to an $O(n^2)$ algorithm. The time complexity can be improved to $O(n \log n)$ by using binary search to find the largest k.

```
1  unsigned divide_x_y(unsigned x, unsigned y) {
2    unsigned res = 0;
3    while (x >= y) {
4      int power = 1;
5      // Checks (y << power) >= (y << (power - 1)) to prevent potential
6      // overflow of unsigned.
7      while ((y << power) >= (y << (power - 1)) && (y << power) <= x) {
8        ++power;
9      }
10
11     res += 1U << (power - 1);
12     x -= y << (power - 1);
13   }
14   return res;
15 }
```

Problem 5.3, pg. 44: *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 5.3: We can use a reductionist approach to solve this problem. We have seen how to convert integers to strings in Solution 7.1 on Page 115; this approach works for any base. Converting from strings is the reverse of this process. Therefore, we can convert base b_1 string s to a variable x of integer type, and then convert x to a base b_2 string ans.For example, if the string is "615", $b_1 = 7$ and $b_2 = 13$, then $x = 306$, and the final result is "1A7".

```
1  string convert_base(const string& s, int b1, int b2) {
2    bool neg = s.front() == '-';
3    int x = 0;
4    for (size_t i = (neg == true ? 1 : 0); i < s.size(); ++i) {
5      x *= b1;
6      x += isdigit(s[i]) ? s[i] - '0' : s[i] - 'A' + 10;
7    }
8
9    string ans;
10   while (x) {
11     int r = x % b2;
12     ans.push_back(r >= 10 ? 'A' + r - 10 : '0' + r);
13     x /= b2;
14   }
```

```
15
16   if (ans.empty()) {  // special case: s is 0.
17     ans.push_back('0');
18   }
19   if (neg) {  // s is a negative number.
20     ans.push_back('-');
21   }
22   reverse(ans.begin(), ans.end());
23   return ans;
24 }
```

The time complexity is $O(n(1 + \log_{b_2} b_1))$, where n is the length of s. The reasoning is as follows. First, we perform n multiply and adds to get x from s. Then we perform $\log_{b_2} x$ multiply and adds to get the result. The value x is upper-bounded by $b_1{}^n$, so $\log_{b_2} b_1{}^n = n \log_{b_2} b_1$.

Problem 5.4, pg. 44: *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?*

Solution 5.4: Basically, we want to produce a random integer in $[0, b - a]$. Let $t = b - a + 1$. We can produce a random integer in $[0, t - 1]$, as follows. Let i be the least integer such that $t \le 2^i$.

If t is a power of 2, say $t = 2^i$, then all we need are i calls to the zero-one valued random number generator—the i bits from the calls encode an i bit integer in $[0, t-1]$, and all such numbers are equally likely; so, we can use this integer.

If t is not a power of 2, the i calls may or may not encode an integer in the range 0 to $t - 1$. 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 $[0, t - 1]$, we try again.

```
1  int uniform_random_a_b(int a, int b) {
2    int t = b - a + 1, res;
3    do {
4      res = 0;
5      for (int i = 0; (1 << i) < t; ++i) {
6        // zero_one_random() is the system-provided random number generator.
7        res = (res << 1) | zero_one_random();
8      }
9    } while (res >= t);
10   return res + a;
11 }
```

The time complexity of such randomized algorithms is usually described in terms of average time. The probability of having to try again is less than $\frac{1}{2}$ since $t > 2^{i-1}$. 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 and consequently the expected running time is $O(1)$. (Note that this is independent of t, i.e., it is not the running time averaged over all inputs.)

Problem 5.5, pg. 44: *Which doors are open after the 500-th person has walked through?*

Solution 5.5: As described on Page 33, analyzing a few concrete examples suggests that, independent of n, door k will be open iff k is a perfect square. The rigorous justification for this is as follows.

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 concrete 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, i.e., in $O(1)$ time.

```
1 bool is_door_open(int i) {
2    double sqrt_i = sqrt(i);
3    int floor_sqrt_i = floor(sqrt_i);
4    return floor_sqrt_i * floor_sqrt_i == i;
5 }
```

Variant 5.5.1: 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 5.6, pg. 44: *Design an efficient algorithm for computing the GCD of two numbers without using multiplication, division or the modulus operators.*

Solution 5.6: The straightforward algorithm is based on the recursion $\text{GCD}(x, y) = (x == y)?x : \text{GCD}(\max(x, y) - \min(x, y), \min(x, y))$. It does not use multiplication, division or modulus, but is very slow—its time complexity is $O(\max(x, y))$, which is exponential in the size of the input. (Expressed in binary, the numbers x and y, require $\lceil \lg x \rceil$ and $\lceil \lg y \rceil$ bits respectively.) As an example, if the input is $x = 2^n$, $y = 2$, the algorithm makes 2^{n-1} recursive calls. (The straightforward algorithm can be improved to linear time complexity, but this entails performing integer division.)

Our solution is also based on 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 time complexity is proportional to the sum of the number of bits in x and y, i.e., $O(\log x + \log y)$.

```
1  long long GCD(long long x, long long y) {
2    if (x == 0) {
3      return y;
4    } else if (y == 0) {
5      return x;
6    } else if (!(x & 1) && !(y & 1)) {  // x and y are even.
7      return GCD(x >> 1, y >> 1) << 1;
8    } else if (!(x & 1) && y & 1) {  // x is even, and y is odd.
9      return GCD(x >> 1, y);
10   } else if (x & 1 && !(y & 1)) {  // x is odd, and y is even.
11     return GCD(x, y >> 1);
12   } else if (x > y) {  // both x and y are odd, and x > y.
13     return GCD(x - y, y);
14   }
15   return GCD(x, y - x);  // both x and y are odd, and x <= y.
16 }
```

Problem 6.1, pg. 46: *Write a function that takes an array A of length n 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 $O(n)$ 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 $A[\texttt{larger} + 1 : n - 1]$.

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 $O(1)$, implying the time complexity is $O(n)$.

The implementation is short but tricky, pay attention to the movements of pointers.

```
1  void dutch_flag_partition(int pivot_index, vector<int>* A) {
2    int pivot = (*A)[pivot_index];
```

```
3    /**
4     * Keep the following invariants during partitioning:
5     * bottom group: (*A)[0 : smaller - 1].
6     * middle group: (*A)[smaller : equal - 1].
7     * unclassified group: (*A)[equal : larger].
8     * top group: (*A)[larger + 1 : A->size() - 1].
9     */
10    int smaller = 0, equal = 0, larger = A->size() - 1;
11    // When there is any unclassified element.
12    while (equal <= larger) {
13      // (*A)[equal] is the incoming unclassified element.
14      if ((*A)[equal] < pivot) {
15        swap((*A)[smaller++], (*A)[equal++]);
16      } else if ((*A)[equal] == pivot) {
17        ++equal;
18      } else {  // (*A)[equal] > pivot.
19        swap((*A)[equal], (*A)[larger--]);
20      }
21    }
22  }
```

ε-**Variant 6.1.1**: Assuming that keys take one of three values, reorder the array so that all objects with the same key appear together. The order of the subarrays is not important. For example, both Figures 6.1(b) and 6.1(c) on Page 47 are valid answers for Figure 6.1(a) on Page 47. Use $O(1)$ additional space and $O(n)$ time.

ε-**Variant 6.1.2**: Given an array A of n objects with keys that takes one of four values, reorder the array so that all objects that have the same key appear together. Use $O(1)$ additional space and $O(n)$ time.

ε-**Variant 6.1.3**: Given an array A of n objects with Boolean-valued keys, reorder the array so that objects that have the key `false` appear first. Use $O(1)$ additional space and $O(n)$ time.

Variant 6.1.4: Given an array A of n objects with Boolean-valued keys, reorder the array so that objects that have the key `false` appear first. The relative ordering of objects with key `true` should not change. Use $O(1)$ additional space and $O()$ time.

Problem 6.2, *pg. 47*: *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 6.2: 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 minimum as we do a sweep through the array. Here is an implementation.

```
1  int find_battery_capacity(const vector<int>& h) {
2    int min_height = numeric_limits<int>::max(), capacity = 0;
3    for (const int &height : h) {
4      capacity = max(capacity, height - min_height);
5      min_height = min(min_height, height);
6    }
7    return capacity;
8  }
```

ϵ-**Variant** 6.2.1: Let A be an array of integers. Find the length of a longest subarray all of whose entries are equal.

Problem 6.3, pg. 47: *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 6.3: 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:

$$S_i^j = A[i] + \max_{i' < i} B_{i'}^j,$$
$$B_i^j = \max_{i' < i} S_{i'}^{j-1} - A[i]$$

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:

```
1  int max_k_pairs_profits(const vector<int>& A, int k) {
2    vector<int> k_sum(k << 1, numeric_limits<int>::min());
3    for (int i = 0; i < A.size(); ++i) {
4      vector<int> pre_k_sum(k_sum);
5      for (int j = 0, sign = -1; j < k_sum.size() && j <= i; ++j, sign *= -1) {
6        int diff = sign * A[i] + (j == 0 ? 0 : pre_k_sum[j - 1]);
7        k_sum[j] = max(diff, pre_k_sum[j]);
8      }
9    }
10   return k_sum.back();  // returns the last selling profits as the answer.
11 }
```

Note that the improved solution to (2.) on the preceding page specialized to $k = 2$ strictly subsumes the solution to (1.) on the previous page.

Surprisingly, (3.) on the preceding page can be solved almost trivially. Define a *locally maximum subarray* of A to be a subarray $A[i : j]$ such that (1.) all elements within the subarray are equal, (2.) if $i > 0$, $A[i] > A[i-1]$, and (3.) if $j < n - 1$, $A[j] > A[j+1]$. A locally minimum subarray is defined similarly. Call an index i a *local minimum* if $A[i]$ is less than or equal to its neighbors, and a *local maximum* if $A[i]$ is greater than or equal to its neighbors. An optimum solution for (3.) on the previous page then is to buy at every local minimum that begins a locally minimum subarray and sell at every local maximum that ends a locally maximum subarray. A local minimum at the end of the array has to be special-cased as is a local maximum at the start of the array. The code below implements this approach.

```
1  int max_profit_unlimited_pairs(const vector<int>& A) {
2    if (A.size() <= 1) {
3      return 0;
4    }
5
6    int profit = 0, buy = A.front();
7    for (int i = 1; i < A.size() - 1; ++i) {
8      if (A[i + 1] < A[i] && A[i - 1] <= A[i]) {  // sell at local maximum.
9        profit += A[i] - buy;
10       buy = A[i + 1];
11     } else if (A[i + 1] >= A[i] && A[i - 1] > A[i]) {  // buy at local minimum
12       buy = A[i];
13     }
14   }
15
16   if (A.back() > buy) {
17     profit += A.back() - buy;
18   }
19   return profit;
20 }
```

The time complexity is $O(n)$ since we spend $O(1)$ per index.

Problem 6.4, pg. 48: *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 6.4: 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 demonstrate that all the subsets are equally likely, we show something stronger, namely that all permutations of size k are equally likely.

Define a sequence of m elements of S with no repetitions to be an m-permutation of S. It is easy to check that the number of m-permutations of a set of n elements 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 of size m are equally likely with this optimization, all m-permutations are not.

```
1  vector<int> offline_sampling(vector<int> A, int k) {
2    default_random_engine gen((random_device())());  // random num generator.
3    for (int i = 0; i < k; ++i) {
4      // Generate a random int in [i, A.size() - 1].
5      uniform_int_distribution<int> dis(i, A.size() - 1);
```

```
6      swap(A[i], A[dis(gen)]);
7    }
8    A.resize(k);
9    return A;
10 }
```

Variant 6.4.1: The `rand()` function in the standard C library returns a uniformly random number in $[0, \text{RAND\_MAX} - 1]$. Does `rand()` mod n generate a number uniformly distributed $[0, n - 1]$?

Problem 6.5, pg. 48: *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 6.5: 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 show that the algorithm works correctly, we use induction on the number of packets that have been read. Specifically, the induction 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.

```
1  vector<int> reservoir_sampling(istringstream* sin, int k) {
2    int x;
3    vector<int> R;
4    // Store the first k elements.
5    for (int i = 0; i < k && *sin >> x; ++i) {
6      R.emplace_back(x);
7    }
8
9    // After the first k elements.
10   int element_num = k;
11   while (*sin >> x) {
12     default_random_engine gen((random_device())());  // random num generator.
13     // Generate a random int in [0, element_num].
14     uniform_int_distribution<int> dis(0, element_num++);
15     int tar = dis(gen);
16     if (tar < k) {
17       R[tar] = x;
18     }
19   }
20   return R;
21 }
```

The time complexity is proportional to the number of elements in the stream, since we spend $O(1)$ time per element. The space complexity is $O(k)$.

Problem 7.1, pg. 49: *Implement string/integer inter-conversion functions. Use the following function signatures:* `String intToString(int x)` *and* `int stringToInt(String s).`

Solution 7.1: 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.

```
1  string intToString(int x) {
2    bool is_negative;
3    if (x < 0) {
4      x = -x, is_negative = true;
5    } else {
6      is_negative = false;
7    }
8
9    string s;
10   while (x) {
11     s.push_back('0' + x % 10);
12     x /= 10;
13   }
14   if (s.empty()) {
15     return {"0"};  // x is 0.
16   }
17
18   if (is_negative) {
```

```
19       s.push_back('-');
20    }
21    reverse(s.begin(), s.end());
22    return s;
23 }
24
25 // We define the valid strings for this function as those matching regexp
26 // -?[0-9]+.
27 int stringToInt(const string& s) {
28    // "-" starts as a valid integer, but has no digits.
29    if (s == "-") {
30      throw invalid_argument("illegal input");
31    }
32
33    bool is_negative = s[0] == '-';
34    int x = 0;
35    for (int i = is_negative; i < s.size(); ++i) {
36      if (isdigit(s[i])) {
37        x = x * 10 + s[i] - '0';
38      } else {
39        throw invalid_argument("illegal input");
40      }
41    }
42    return is_negative ? -x : x;
43 }
```

Problem 7.2, pg. 49: *Implement a function for reversing the words in a string s. Your function should use O(1) space.*

Solution 7.2: 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".

```
1  void reverse_words(string* s) {
2    // Reverse the whole string first.
3    reverse(s->begin(), s->end());
4
5    size_t start = 0, end;
6    while ((end = s->find(" ", start)) != string::npos) {
7      // Reverse each word in the string.
8      reverse(s->begin() + start, s->begin() + end);
9      start = end + 1;
10   }
11   // Reverse the last word.
12   reverse(s->begin() + start, s->end());
13 }
```

Since we spend $O(1)$ per character, the time complexity is $O(n)$, where n is the length of s.

Problem 7.3, pg. 50: *Write a function which takes as input a phone number, specified as a string of digits, return all possible character sequences that correspond to the phone number. The cell phone keypad is specified by a mapping M that takes a digit and returns the corresponding set of characters. The character sequences do not have to be legal words or phrases.*

Solution 7.3: 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 $x \in M(P[k])$.

```
1  void phone_mnemonic(const string &num) {
2    string ans(num.size(), 0);
3    phone_mnemonic_helper(num, 0, &ans);
4  }
5
6  const int kNumTelDigits = 10;
7
8  const array<string, kNumTelDigits> M = {{"0", "1", "ABC", "DEF", "GHI",
9                                           "JKL", "MNO", "PQRS", "TUV",
10                                          "WXYZ"}};
11
12 void phone_mnemonic_helper(const string &num, int d, string* ans) {
13   if (d == num.size()) {  // get enough characters and output answer.
14     cout << *ans << endl;
15   } else {
16     for (const char &c : M[num[d] - '0']) {  // try all combinations.
17       (*ans)[d] = c;
18       phone_mnemonic_helper(num, d + 1, ans);
19     }
20   }
21 }
```

Since there are no more than 4 possible characters for each digit, the number of recursive calls $T(n)$ satisfies $T(n) \leq 4T(n - 1)$, which solves to $T(n) = O(4^n)$. For the function calls that entail recursion, the time spent within the function, not including the recursive calls, is $O(1)$. For the base case, printing a sequence of length n takes time $O(n)$. Therefore, the time complexity is $O(4^n n)$.

Variant 7.3.1: Solve the same problem without using recursion.

Problem 8.1, pg. 52: *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 8.3 on Page 52. The only field you can change in a node is next.*

Solution 8.1: 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.

```
1  shared_ptr<ListNode<int>> merge_sorted_linked_lists(
2       shared_ptr<ListNode<int>> F, shared_ptr<ListNode<int>> L) {
3    shared_ptr<ListNode<int>> sorted_head = nullptr, tail = nullptr;
4
5    while (F && L) {
6      append_node_and_advance(&sorted_head, &tail, F->data < L->data ? &F : &L);
7    }
8
9    // Appends the remaining nodes of F.
10   if (F) {
11     append_node(F, &sorted_head, &tail);
12   }
13   // Appends the remaining nodes of L.
14   if (L) {
15     append_node(L, &sorted_head, &tail);
16   }
17   return sorted_head;
18 }
19
20 void append_node_and_advance(shared_ptr<ListNode<int>>* head,
21                              shared_ptr<ListNode<int>>* tail,
22                              shared_ptr<ListNode<int>>* node) {
23   append_node(*node, head, tail);
24   *node = (*node)->next;  // advances node.
25 }
26
27 void append_node(const shared_ptr<ListNode<int>>& node,
28                  shared_ptr<ListNode<int>>* head,
29                  shared_ptr<ListNode<int>>* tail) {
30   *head ? (*tail)->next = node : *head = node;
31   *tail = node;  // resets tail to the last node.
32 }
```

The worst case, from a runtime perspective, corresponds to the case when the lists
are of comparable length, so the time complexity is $O(n_L + n_F)$, where n_L and n_F are
the lengths of lists L and F, respectively. (In the best case, one list is much shorter
than the other and all its entries appear at the beginning of the merged list.)

ε-**Variant** 8.1.1: Solve the same problem when the lists are doubly linked.

Problem 8.2, *pg.* 52: *Give a linear time nonrecursive function that reverses a singly linked
list. The function should use no more than constant storage beyond that needed for the list
itself.*

Solution 8.2: The natural way of implementing the reversal is through recursion.
This approach has (*n*) time complexity, since $O(1)$ time is spent within each call, and
one node is moved to its correct location.

```
1  shared_ptr<ListNode<int>> reverse_linked_list(
2      const shared_ptr<ListNode<int>>& head) {
3    if (!head || !head->next) {
4      return head;
5    }
6
7    shared_ptr<ListNode<int>> new_head = reverse_linked_list(head->next);
8    head->next->next = head;
9    head->next = nullptr;
10   return new_head;
11 }
```

However, the recursive approach implicitly uses $O(n)$ space on the stack. The function is not tail recursive, which precludes compilers from automatically converting the function to an iterative one.

Consider the following (nonrecursive) solution: traverse the list with two pointers, and update the trailing pointer's next field. It uses $O(1)$ additional storage, and has (n) time complexity.

```
1  shared_ptr<ListNode<int>> reverse_linked_list(
2      const shared_ptr<ListNode<int>>& head) {
3    shared_ptr<ListNode<int>> prev = nullptr, curr = head;
4    while (curr) {
5      shared_ptr<ListNode<int>> temp = curr->next;
6      curr->next = prev;
7      prev = curr;
8      curr = temp;
9    }
10   return prev;
11 }
```

Problem 8.3, pg. 52: *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 8.3: 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 $O(n)$ 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 LSBs in the next pointer are 0. Bit fiddling can be used to set the LSB 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 8.2 on Page 118. 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 traverse the list in two loops—the outer loop traverses the nodes one-by-one, and the inner loop starts from the head, and traverses m nodes, where m is the number of nodes traversed 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 traverse the list. In each iteration, advance `slow` by one and `fast` by two. The list has a cycle if and only if the two pointers meet. The reasoning is as follows. 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 $2(i - F)$ mod C. If the slow pointer reaches the first node at iteration S, at iteration $i \geq S$, it will be at node $(i - S)$ mod C. The difference between the pointer locations after the slow pointer reaches the first node in the cycle is $2(i - F) - (i - S)$ mod $C = i - (2F - S)$ mod C. As i increases by one in each iteration, the equation $(i - (2F - S))$ mod $C = 0$ must have 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 slow pointer, and counting the number of times we have to advance the fast pointer to come back to the slow 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:

```
1  shared_ptr<ListNode<int>> has_cycle(const shared_ptr<ListNode<int>>& head) {
2    shared_ptr<ListNode<int>> fast = head, slow = head;
3
4    while (slow && slow->next && fast && fast->next && fast->next->next) {
5      slow = slow->next, fast = fast->next->next;
6      if (slow == fast) {  // there is a cycle.
7        // Calculates the cycle length.
8        int cycle_len = 0;
9        do {
10          ++cycle_len;
11          fast = fast->next;
12        } while (slow != fast);
13
14        // Tries to find the start of the cycle.
15        slow = head, fast = head;
16        // Fast pointer advances cycle_len first.
17        while (cycle_len--) {
18          fast = fast->next;
```

```
19          }
20          // Both pointers advance at the same time.
21          while (slow != fast) {
22            slow = slow->next, fast = fast->next;
23          }
24          return slow;  // the start of cycle.
25        }
26    }
27    return nullptr;  // no cycle.
28  }
```

Its time complexity is $O(F) + O(C) = O(n)$—$O(F)$ for both pointers to reach the cycle, and $O(C)$ for them to overlap once the slower one enters the cycle.

ϵ-**Variant 8.3.1**: 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?

```
1 shared_ptr<ListNode<int>> has_cycle(const shared_ptr<ListNode<int>>& head) {
2   shared_ptr<ListNode<int>> fast = head, slow = head;
3
4   while (slow && slow->next && fast && fast->next && fast->next->next) {
5     slow = slow->next, fast = fast->next->next;
6     if (slow == fast) {  // there is a cycle.
7       // Tries to find the start of the cycle.
8       slow = head;
9       // Both pointers advance at the same time.
10      while (slow != fast) {
11        slow = slow->next, fast = fast->next;
12      }
13      return slow;  // slow is the start of cycle.
14    }
15  }
16  return nullptr;  // means no cycle.
17 }
```

Problem 8.4, pg. 53: *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 8.4: 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 to on this page.

(a) Initial list.

(b) After Stage (1.).

(c) After Stage (2.).

(d) After Stage (3.).
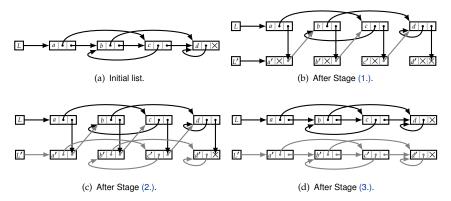
Figure 21.1: Duplicating a postings list.

Code implementing the copy is given below.

```
1  shared_ptr<ListNode<int>> copy_postings_list(
2      const shared_ptr<ListNode<int>>& L) {
3    // Returns empty list if L is nullptr.
4    if (!L) {
5      return nullptr;
6    }
7
8    // 1st stage: Copies the nodes from L.
9    shared_ptr<ListNode<int>> p = L;
10   while (p) {
11     auto temp =
12         make_shared<ListNode<int>>(ListNode<int>{p->data, p->next, nullptr});
13     p->next = temp;
14     p = temp->next;
15   }
16
17   // 2nd stage: Updates the jump field.
18   p = L;
19   while (p) {
20     if (p->jump) {
21       p->next->jump = p->jump->next;
22     }
23     p = p->next->next;
24   }
25
26   // 3rd stage: Restores the next field.
27   p = L;
```

```
28    shared_ptr<ListNode<int>> copied = p->next;
29    while (p->next) {
30      shared_ptr<ListNode<int>> temp = p->next;
31      p->next = temp->next;
32      p = temp;
33    }
34    return copied;
35 }
```

The time complexity is $O(n)$, where n is the length of the list. The space complexity is $O(1)$, beyond the space allocated for the result.

Problem 9.1, pg. 54: *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. If the stack contains n elements, you can use $O(n)$ space, in addition to what is required for the elements themselves.*

Solution 9.1: 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.

```
1 class Stack {
2  public:
3    bool empty() const { return s_.empty(); }
4
5    int max() const {
6      if (!empty()) {
7        return s_.top().second;
8      }
9      throw length_error("empty stack");
10   }
11
12   int pop() {
13     if (empty()) {
14       throw length_error("empty stack");
15     }
16     int ret = s_.top().first;
17     s_.pop();
18     return ret;
19   }
20
21   void push(int x) {
22     s_.emplace(x, std::max(x, empty() ? x : s_.top().second));
23   }
24
25  private:
```

```
26      stack<pair<int, int>> s_;
27 };
```

Each of the specified methods has time complexity $O(1)$. The additional space complexity is $O(n)$, regardless of the stored keys.

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.

- k is smaller than m. The auxiliary stack is not updated.
- k is equal to m. We increment the second component of the pair stored at the top of the auxiliary stack.
- 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.

- k is less than m. The auxiliary stack is not updated.
- 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 .

```
1  class Stack {
2   public:
3    bool empty() const { return s_.empty(); }
4
5    int max() const {
6      if (!empty()) {
7        return aux_.top().first;
8      }
9      throw length_error("empty stack");
10   }
11
12   int pop() {
13     if (empty()) {
14       throw length_error("empty stack");
15     }
16     int ret = s_.top();
17     s_.pop();
18     if (ret == aux_.top().first) {
19       --aux_.top().second;
20       if (aux_.top().second == 0) {
21         aux_.pop();
22       }
23     }
24     return ret;
25   }
26
27   void push(int x) {
```

```
28      s_.emplace(x);
29      if (!aux_.empty()) {
30        if (x == aux_.top().first) {
31          ++aux_.top().second;
32        } else if (x > aux_.top().first) {
33          aux_.emplace(x, 1);
34        }
35      } else {
36        aux_.emplace(x, 1);
37      }
38    }
39
40  private:
41    stack<int> s_;
42    stack<pair<int, int>> aux_;
43 };
```

The worst-case additional space complexity is $O(n)$, 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 $O(1)$.
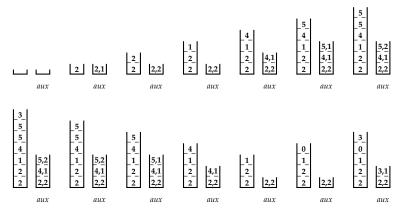
Figure 21.2: The primary and auxiliary stacks for the following operations: push(2), push(2), push(1), push(4), push(5), push(5), push(3), pop(), pop(), pop(), pop(), push(0), push(3). Both stacks are initially empty, and their progression is shown from left-to-right, then top-to-bottom. The top of the 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*.

Problem 9.2, pg. 55: *Given the root node r of a binary tree, print all the keys at r and its descendants. The keys should be printed in the order of the corresponding nodes' depths. Specifically, all keys corresponding to nodes of depth d should appear in a single line, and the next line should correspond to keys corresponding to nodes of depth d + 1. You cannot use recursion. You may use a single queue, and constant additional storage. For example, you should print*

314

6 6

271 561 2 271

28 0 3 1 28

17 401 257

641

for the binary tree in Figure 10.1 on Page 57.

Solution 9.2: We maintain a queue of nodes to process. Specifically the queue contains nodes at depth l followed by nodes at depth $l + 1$. After all nodes at depth l are processed, the head of the queue is a node at depth $l + 1$; processing this node introduces nodes from depth $l + 2$ to the end of the queue. We use a count variable that records the number of nodes at the depth of the head of the queue that remain to be processed. When all nodes at depth l are processed, the queue consists of exactly the set of nodes at depth $l + 1$, and count is updated to the size of the queue.

```
1  void print_binary_tree_depth_order(const unique_ptr<BinaryTreeNode<int>>& r) {
2    // Prevents empty tree.
3    if (!r) {
4      return;
5    }
6
7    queue<BinaryTreeNode<int>*> q;
8    q.emplace(r.get());
9    size_t count = q.size();
10   while (!q.empty()) {
11     cout << q.front()->data << ' ';
12     if (q.front()->left) {
13       q.emplace(q.front()->left.get());
14     }
15     if (q.front()->right) {
16       q.emplace(q.front()->right.get());
17     }
18     q.pop();
19     if (--count == 0) {  // Finish printing nodes in the current depth.
20       cout << endl;
21       count = q.size();
22     }
23   }
24 }
```

Since each node is enqueued and dequeued exactly once, the time complexity is $O(n)$ where n is the number of nodes in the tree. The space complexity is $O(m)$, where m is the maximum number of nodes at a specific depth. This could be as high as $O(n)$, e.g., for a perfect binary tree (Page 58).

ϵ-**Variant 9.2.1:** Write a function which takes as input a binary tree where each node is labeled with an integer and prints all the node keys in top down, alternating left-to-right and right-to-left order, starting from left-to-right. For example, you should print

314
6 6
271 561 2 271
28 1 3 0 28
17 401 257
641
for the binary tree in Figure 10.1 on Page 57.

ϵ-**Variant 9.2.2**: Write a function which takes as input a binary tree where each node
is labeled with an integer and prints all the node keys in a bottom up, left-to-right
order. For example, if the input is the tree in Figure 10.1 on Page 57, your function
should print   641
17 401 257
28 0 3 1 28
271 561 2 271
6 6
314

Problem 9.3, pg. 55: *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 9.3: 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.

```
1  class Queue {
2   public:
3    explicit Queue(size_t cap) : data_(cap) {}
4
5    void enqueue(int x) {
6      // Dynamically resizes due to data_.size() limit.
7      if (count_ == data_.size()) {
8        // Rearranges elements.
9        rotate(data_.begin(), data_.begin() + head_, data_.end());
10       head_ = 0, tail_ = count_;  // resets head and tail.
11       data_.resize(data_.size() << 1);
12     }
13     // Performs enqueue.
14     data_[tail_] = x;
15     tail_ = (tail_ + 1) % data_.size(), ++count_;
16   }
```

```
17
18   int dequeue() {
19     if (!count_) {
20       throw length_error("empty queue");
21     }
22     --count_;
23     int ret = data_[head_];
24     head_ = (head_ + 1) % data_.size();
25     return ret;
26   }
27
28   size_t size() const { return count_; }
29
30   private:
31   size_t head_ = 0, tail_ = 0, count_ = 0;
32   vector<int> data_;
33 };
```

The time complexity of each operation is $O(1)$.

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 10.1, pg. 59: *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 10.1: 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 and $O(n)$ time, where n is the number of nodes of the tree.

We can 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. This 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 traversal 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. The time complexity is the same as that for a postorder traversal, namely $O(n)$.

```
1 bool is_balanced_binary_tree(const unique_ptr<BinaryTreeNode<int>>& T) {
```

```
2    return get_height(T) != -2;
3  }
4
5  int get_height(const unique_ptr<BinaryTreeNode<int>>& T) {
6    if (!T) {
7      return -1;  // base case.
8    }
9
10   int l_height = get_height(T->left);
11   if (l_height == -2) {
12     return -2;  // left subtree is not balanced.
13   }
14   int r_height = get_height(T->right);
15   if (r_height == -2) {
16     return -2;  // right subtree is not balanced.
17   }
18
19   if (abs(l_height - r_height) > 1) {
20     return -2;  // current node T is not balanced.
21   }
22   return max(l_height, r_height) + 1;  // return the height.
23 }
```

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 so, if the stack height ever exceeds h_n, we return -2.

Variant **10.1**.**1**: Write a function that returns the size of the largest subtree that is complete.

Problem **10.2**, **pg.** **60**: *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 **10.2**: 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.

```
1  BinaryTreeNode<int>* LCA(const unique_ptr<BinaryTreeNode<int>>& T,
2                           const unique_ptr<BinaryTreeNode<int>>& a,
3                           const unique_ptr<BinaryTreeNode<int>>& b) {
4    if (!T) {  // empty subtree.
5      return nullptr;
6    } else if (T == a || T == b) {
7      return T.get();
8    }
```

```
9
10    auto* l_res = LCA(T->left, a, b), *r_res = LCA(T->right, a, b);
11    if (l_res && r_res) {
12      return T.get();  // found a and b in different subtrees.
13    } else {
14      return l_res ? l_res : r_res;
15    }
16 }
```

The algorithm is structurally similar to a recursive inorder traversal, and the complexities are the same.

Problem 10.3, *pg. 60*: *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 10.3: The standard idiom for an inorder traversal is traverse-left, visit-root, traverse-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 T if the child we completed traversing was T's left child (in which case we need to traverse T and then T's right child) or a right child (in which case we have completed traversing T). We achieve this by storing the child in a prev variable before we move to the parent, T. We then compare prev with T's left child and the right child.

```
1  void inorder_traversal(const unique_ptr<BinaryTreeNode<int>>& T) {
2    // Empty tree.
3    if (!T) {
4      return;
5    }
6
7    BinaryTreeNode<int>* prev = nullptr, *curr = T.get(), *next;
8    while (curr) {
9      if (!prev || prev->left.get() == curr || prev->right.get() == curr) {
10        if (curr->left) {
11          next = curr->left.get();
12        } else {
13          cout << curr->data << endl;
14          next = (curr->right ? curr->right.get() : curr->parent);
15        }
16      } else if (curr->left.get() == prev) {
17        cout << curr->data << endl;
18        next = (curr->right ? curr->right.get() : curr->parent);
19      } else {  // curr->right.get() == prev.
20        next = curr->parent;
21      }
22
23      prev = curr;
24      curr = next;
25    }
```

```
26  }
```

The time complexity is $O(n)$, where n is the number of nodes in T.

ϵ-**Variant 10.3**.1: How would you perform preorder and postorder traversals iteratively using $O(1)$ additional space? Your algorithm cannot modify the tree. Nodes have an explicit parent field.

Problem 10.4, pg. 60: *Design an algorithm that takes a node n in a binary tree, and returns its successor. Assume that each node has a* **parent** *field; the* **parent** *field of root is* **null**.

Solution 10.4: If n has a nonempty right subtree, we return the leftmost node in the right subtree. If n does not have a right child, then we keep traversing parent pointers till we encounter a node which is the left child of its parent, in which case that parent is n's successor. If we reach the root then n is the last node in the inorder traversal and has no successor.

```
1  BinaryTreeNode<int>* find_successor(
2      const unique_ptr<BinaryTreeNode<int>>& node) {
3    auto* n = node.get();
4    if (n->right) {
5      // Find the leftmost element in n's right subtree.
6      n = n->right.get();
7      while (n->left) {
8        n = n->left.get();
9      }
10     return n;
11   }
12
13   // Find the first parent whose left child contains n.
14   while (n->parent && n->parent->right.get() == n) {
15     n = n->parent;
16   }
17   // Return nullptr means n does not have successor.
18   return n->parent;
19 }
```

Since the number of edges followed cannot be more than the tree height, the time complexity is $O(h)$, where h is the height of the tree.

Problem 11.1, pg. 62: *Design an algorithm that takes a set of files containing stock trades sorted by increasing trade times, and writes a single file containing the trades appearing in the individual files sorted in the same order. The algorithm should use very little RAM, ideally of the order of a few kilobytes.*

Solution 11.1: In the abstract, we are trying to merge k sequences sorted in increasing order. One way to do this is to repeatedly pick the smallest element amongst the smallest remaining elements of each of the k sequences. A min-heap is ideal for maintaining a set of elements when we need to insert arbitrary values, as well as query for the smallest element. There are no more than k elements in the min-heap. 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. The space complexity is $O(k)$ beyond the space needed to write the final result. The implementation is given below. Note that for each element we need to store the sequence it came from. For ease of exposition, we show how to merge sorted arrays, rather than files. The only difference is that for the file case we do not need to explicitly maintain an index for next unprocessed element in each sequence—the file I/O library tracks the first unread entry in the file.

```
1  struct Compare {
2    bool operator()(const pair<int, int>& lhs, const pair<int, int>& rhs) {
3      return lhs.first > rhs.first;
4    }
5  };
6
7  vector<int> merge_arrays(const vector<vector<int>>& S) {
8    priority_queue<pair<int, int>, vector<pair<int, int>>, Compare> min_heap;
9    vector<int> S_idx(S.size(), 0);
10
11   // Every array in S puts its smallest element in heap.
12   for (int i = 0; i < S.size(); ++i) {
13     if (S[i].size() > 0) {
14       min_heap.emplace(S[i][0], i);
15       S_idx[i] = 1;
16     }
17   }
18
19   vector<int> ret;
20   while (!min_heap.empty()) {
21     pair<int, int> p = min_heap.top();
22     ret.emplace_back(p.first);
23     // Add the smallest element into heap if possible.
24     if (S_idx[p.second] < S[p.second].size()) {
25       min_heap.emplace(S[p.second][S_idx[p.second]++], p.second);
26     }
27     min_heap.pop();
28   }
29   return ret;
30 }
```

Problem 11.2, pg. 63: *How would you compute the k stars which are closest to the Earth? You have only a few megabytes of RAM.*

Solution 11.2: If RAM was not a limitation, we could read the data into an array, and compute the *k* smallest elements using a selection algorithm.

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 $O(k)$.

```cpp
1  class Star {
2   public:
3    // The distance between this star to the Earth.
4    double distance() const { return sqrt(x_ * x_ + y_ * y_ + z_ * z_); }
5
6    bool operator<(const Star& s) const { return distance() < s.distance(); }
7
8    int ID_;
9    double x_, y_, z_;
10 };
11
12 vector<Star> find_closest_k_stars(int k, istringstream *sin) {
13   // Use max_heap to find the closest k stars.
14   priority_queue<Star, vector<Star>> max_heap;
15   string line;
16
17   // Record the first k stars.
18   while (getline(*sin, line)) {
19     stringstream line_stream(line);
20     string buf;
21     getline(line_stream, buf, ',');
22     int ID = stoi(buf);
23     array<double, 3> data;  // stores x, y, and z.
24     for (int i = 0; i < 3; ++i) {
25       getline(line_stream, buf, ',');
26       data[i] = stod(buf);
27     }
28     Star s{ID, data[0], data[1], data[2]};
29
30     if (max_heap.size() == k) {
31       // Compare the top of heap with the incoming star.
32       Star far_star = max_heap.top();
33       if (s < far_star) {
34         max_heap.pop();
35         max_heap.emplace(s);
36       }
37     } else {
38       max_heap.emplace(s);
39     }
40   }
41
42   // Store the closest k stars.
43   vector<Star> closest_stars;
44   while (!max_heap.empty()) {
45     closest_stars.emplace_back(max_heap.top());
46     max_heap.pop();
47   }
48   return closest_stars;
```

```
49 }
```

Variant 11.2.1: 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. What are the worst case inputs for your algorithm?

Problem 11.3, pg. 63: *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 11.3: 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:

```
1  void online_median(istringstream* sin) {
2    // Min-heap stores the bigger part of the stream.
3    priority_queue<int, vector<int>, greater<int>> H;
4    // Max-heap stores the smaller part of the stream.
5    priority_queue<int, vector<int>, less<int>> L;
6
7    int x;
8    while (*sin >> x) {
9      if (!L.empty() && x > L.top()) {
10       H.emplace(x);
11     } else {
12       L.emplace(x);
13     }
14     if (H.size() > L.size() + 1) {
15       L.emplace(H.top());
16       H.pop();
17     } else if (L.size() > H.size() + 1) {
18       H.emplace(L.top());
19       L.pop();
20     }
21
22     if (H.size() == L.size()) {
23       cout << 0.5 * (H.top() + L.top()) << endl;
24     } else {
25       cout << (H.size() > L.size() ? H.top() : L.top()) << endl;
26     }
27   }
28 }
```

The time complexity per entry is $O(\log n)$, corresponding to insertion and extraction from a heap.

Problem 12.1, pg. 66: *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 12.1 on Page 66 your algorithm should return 3 if k = 108; if k = 285, your algorithm should return 6.*

Solution 12.1: 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.

```
1  int search_first(const vector<int>& A, int k) {
2    int l = 0, r = A.size() - 1, res = -1;
3    while (l <= r) {
4      int m = l + ((r - l) >> 1);
5      if (A[m] > k) {
6        r = m - 1;
7      } else if (A[m] == k) {
8        // Record the solution and keep searching the left part.
9        res = m, r = m - 1;
10     } else {  // A[m] < k
11       l = m + 1;
12     }
13   }
14   return res;
15 }
```

The complexity bound is still $O(\log n)$—this is because each iteration reduces the size of the subarray being searched by half.

ϵ-**Variant 12.1.1**: 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 12.1.2**: 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 12.2, pg. 66: *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 12.2 on Page 67, your algorithm should return 4.*

Solution 12.2: 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.

```
1  int search_smallest(const vector<int>& A) {
2    int l = 0, r = A.size() - 1;
3    while (l < r) {
4      int m = l + ((r - l) >> 1);
5      if (A[m] > A[r]) {
6        l = m + 1;
7      } else {  // A[m] <= A[r].
8        r = m;
9      }
10   }
11   return l;
12 }
```

The time complexity is the same as that of binary search, namely $O(\log n)$.

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 code for the scenario when elements may be repeated:

```
1  int search_smallest(const vector<int>& A) {
2    return search_smallest_helper(A, 0, A.size() - 1);
3  }
4
5  int search_smallest_helper(const vector<int>& A, int l, int r) {
6    if (l == r) {
7      return l;
8    }
9
10   int m = l + ((r - l) >> 1);
11   if (A[m] > A[r]) {
12     return search_smallest_helper(A, m + 1, r);
13   } else if (A[m] < A[r]) {
14     return search_smallest_helper(A, l, m);
15   } else {  // A[m] == A[r].
16     // Smallest element must exist in either left or right side.
17     int l_res = search_smallest_helper(A, l, m);
18     int r_res = search_smallest_helper(A, m + 1, r);
19     return A[r_res] < A[l_res] ? r_res : l_res;
20   }
21 }
```

Variant 12.2.1: Design an $O(\log n)$ algorithm for finding the position of an element k in a cyclically sorted array.

Problem 12.3, pg. 67: *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 12.3: 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 $[b, t]$ that contains x, and use binary search to iteratively half the size of the interval. Perform the iteration while $b < t$. At each iteration set $x = b + \lfloor \frac{t-b}{2} \rfloor$. If $A[x] < B[(k - x) - 1]$, then $A[x]$ must be in the first k elements of the union, so we update b to $x + 1$ and continue. Similarly, if $A[x-1] > B[k-x]$, then $A[x-1]$ cannot be in the first k elements, so we can update t to $x - 1$. Otherwise, we must have $B[(k - x) - 1] \leq A[x]$ and $A[x - 1] \leq B[k - x]$, in which case the result is the larger of $A[x - 1]$ and $B[(k - x) - 1]$, since the first x elements of A and the first $k - x$ elements of B when sorted end in $A[x - 1]$ or $B[(k - x) - 1]$.

If the iteration ends without returning, it must be that $b = t$. Clearly, $x = b = t$. We simply return the larger of $A[x - 1]$ and $B[(k - x) - 1]$. (If $A[x - 1] = B[(k - x) - 1]$, we arbitrarily return either.)

The initial values for b and t need to be chosen carefully. Naïvely setting $b = 0, t = k$ does not work, since this choice may lead to array indices in the search lying outside the range of valid indices. The indexing constraints for A and B can be resolved by initializing b to $\max(0, k - n)$ and t to $\min(m, k)$.

```
1  int find_kth_in_two_sorted_arrays(const vector<int>& A, const vector<int>& B,
2                                     int k) {
3    // Lower bound of elements we will choose in A.
4    int b = max(0, static_cast<int>(k - B.size()));
5    // Upper bound of elements we will choose in A.
6    int t = min(static_cast<int>(A.size()), k);
7
8    while (b < t) {
9      int x = b + ((t - b) >> 1);
10     int A_x_1 = (x <= 0 ? numeric_limits<int>::min() : A[x - 1]);
11     int A_x = (x >= A.size() ? numeric_limits<int>::max() : A[x]);
12     int B_k_x_1 = (k - x <= 0 ? numeric_limits<int>::min() : B[k - x - 1]);
13     int B_k_x = (k - x >= B.size() ? numeric_limits<int>::max() : B[k - x]);
14
15     if (A_x < B_k_x_1) {
16       b = x + 1;
17     } else if (A_x_1 > B_k_x) {
18       t = x - 1;
19     } else {
20       // B[k - x - 1] <= A[x] && A[x - 1] < B[k - x].
21       return max(A_x_1, B_k_x_1);
22     }
23   }
24
25   int A_b_1 = b <= 0 ? numeric_limits<int>::min() : A[b - 1];
26   int B_k_b_1 = k - b - 1 < 0 ? numeric_limits<int>::min() : B[k - b - 1];
27   return max(A_b_1, B_k_b_1);
28 }
```

Since in each iteration we halve the length of $[b, t]$ the time complexity is $O(\log n)$.

Problem 12.4, pg. 67: *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 12.4: 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 MSBs 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.

```
1  int find_missing_element(ifstream* ifs) {
2    vector<size_t> counter(1 << 16, 0);
3    unsigned int x;
4    while (*ifs >> x) {
5      ++counter[x >> 16];
6    }
7
8    for (int i = 0; i < counter.size(); ++i) {
9      // Find one bucket contains less than (1 << 16) elements.
10     if (counter[i] < (1 << 16)) {
11       bitset<(1 << 16)> bit_vec;
12       ifs->clear();
13       ifs->seekg(0, ios::beg);
14       while (*ifs >> x) {
15         if (i == (x >> 16)) {
16           bit_vec.set(((1 << 16) - 1) & x);  // gets the lower 16 bits of x.
17         }
18       }
19       ifs->close();
20
21       for (int j = 0; j < (1 << 16); ++j) {
22         if (bit_vec.test(j) == 0) {
23           return (i << 16) | j;
24         }
25       }
26     }
27   }
28 }
```

Problem 13.1, pg. 69: *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 13.1: 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. Specifically, we store the sorted strings as keys. For each key, the value is an array of the corresponding strings.

```
1  void find_anagrams(const vector<string>& dictionary) {
2    // Get the sorted string and then insert into hash table.
3    unordered_map<string, vector<string>> hash;
4    for (const string& s : dictionary) {
5      string sorted_str(s);
6      // Use sorted string as the hash code.
7      sort(sorted_str.begin(), sorted_str.end());
8      hash[sorted_str].emplace_back(s);
9    }
10
11   for (const pair<string, vector<string>>& p : hash) {
12     // Multiple strings with the same hash code => anagrams.
13     if (p.second.size() >= 2) {
14       // Output all strings.
15       for (const auto& s : p.second) {
16         cout << s << " ";
17       }
18       cout << endl;
19     }
20   }
21 }
```

Let the maximum string length be *m*, and the total number of strings be *n*. The computation consists of *n* calls to sort and *n* insertions into the hash table, followed by an iteration over the keys and printing corresponding values. Sorting the keys has time complexity $O(nm \log m)$; the insertion and printing has time complexity $O(nm)$, yielding an $O(nm \log m)$ time complexity overall. In principle, we can use counting sort to reduce the time to sort to $O(nm)$. For small strings ($m \leq 100$) the initialization overhead of counting sort is too high to see a benefit.

Problem 13.2, *pg. 69*: *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 13.2: Assume the string encoding the magazine is as long as the string encoding the letter. (If not, the answer is `false`.) 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`.

```
1  bool anonymous_letter(const string& L, const string& M) {
2    unordered_map<char, int> hash;
3    // Insert all chars in L into a hash table.
4    for_each(L.begin(), L.end(), [&hash](const char & c) { ++hash[c]; });
5
6    // Check chars in M that could cover chars in a hash table.
7    for (const char& c : M) {
```

```
8      auto it = hash.find(c);
9      if (it != hash.cend()) {
10       if (--it->second == 0) {
11         hash.erase(it);
12         if (hash.empty() == true) {
13           return true;
14         }
15       }
16     }
17   }
18   // No entry in hash means L can be covered by M.
19   return hash.empty();
20 }
```

In the worst case, the letter is not constructible or the last character of M is essentially required. Therefore, the time complexity is $O(n_M)$ where n_M is the length of M The space complexity is $O(c_L)$, where c_L is the number of distinct characters appearing L.

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 13.3, pg. 70: *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 13.3: 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 $n(n-1)/2$ 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. There are at most $n(n-1)/2$ lines, so the iteration takes $O(n^2)$ time, yielding an overall time bound of $O(n^2)$.

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. If the generated uniform $[0, 1]$ random number lies into $[0.3, 0.6]$ we return the number 6.

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.

```cpp
1  struct Point {
2    // Equal function for hash.
3    bool operator==(const Point& that) const {
4      return x == that.x && y == that.y;
5    }
6
7    int x, y;
8  };
9
10 // Hash function for Point.
11 struct HashPoint {
12   size_t operator()(const Point& p) const {
13     return hash<int>()(p.x) ^ hash<int>()(p.y);
14   }
15 };
16
17 pair<int, int> get_canonical_fractional(int a, int b) {
18   int gcd = GCD(abs(a), abs(b));
19   a /= gcd, b /= gcd;
20   return b < 0 ? make_pair(-a, -b) : make_pair(a, b);
21 }
22
23 // Line function of two points, a and b, and the equation is
24 // y = x(b.y - a.y) / (b.x - a.x) + (b.x * a.y - a.x * b.y) / (b.x - a.x).
25 struct Line {
26   Line(const Point& a, const Point& b)
27       : slope(a.x != b.x ? get_canonical_fractional(b.y - a.y, b.x - a.x)
28                          : make_pair(1, 0)),
29         intercept(a.x != b.x ? get_canonical_fractional(b.x * a.y - a.x * b.y,
30                                                         b.x - a.x)
31                             : make_pair(a.x, 1)) {}
32
33   // Equal function for Line.
34   bool operator==(const Line& that) const {
35     return slope == that.slope && intercept == that.intercept;
36   }
37
38   // Store the numerator and denominator pair of slope unless the line is
39   // parallel to y-axis that we store 1/0.
40   pair<int, int> slope;
41   // Store the numerator and denominator pair of the y-intercept unless
42   // the line is parallel to y-axis that we store the x-intercept.
43   pair<int, int> intercept;
44 };
45
46 // Hash function for Line.
47 struct HashLine {
48   size_t operator()(const Line& l) const {
49     return hash<int>()(l.slope.first) ^ hash<int>()(l.slope.second) ^
```

```
50                hash<int>()(l.intercept.first) ^ hash<int>()(l.intercept.second);
51    }
52  };
53
54  Line find_line_with_most_points(const vector<Point>& P) {
55    // Add all possible lines into hash table.
56    unordered_map<Line, unordered_set<Point, HashPoint>, HashLine> table;
57    for (int i = 0; i < P.size(); ++i) {
58      for (int j = i + 1; j < P.size(); ++j) {
59        Line l(P[i], P[j]);
60        table[l].emplace(P[i]), table[l].emplace(P[j]);
61      }
62    }
63
64    // Return the line with most points have passed.
65    return max_element(table.cbegin(),
66                       table.cend(),
67                       [](const pair<Line, unordered_set<Point, HashPoint>>& a,
68                          const pair<Line, unordered_set<Point, HashPoint>>& b)
69                       { return a.second.size() < b.second.size(); })->first;
70  }
```

Problem 14.1, pg. 72: *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 14.1: The brute-force algorithm is a "loop join", i.e., traversing 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:

```
1   vector<int> intersect_arrs1(const vector<int>& A, const vector<int>& B) {
2     vector<int> intersect;
3     for (int i = 0; i < A.size(); ++i) {
4       if (i == 0 || A[i] != A[i - 1]) {
5         for (int j = 0; j < B.size(); ++j) {
6           if (A[i] == B[j]) {
7             intersect.emplace_back(A[i]);
8             break;
9           }
10        }
11      }
12    }
13    return intersect;
14  }
```

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*.

```
1   vector<int> intersect_arrs2(const vector<int>& A, const vector<int>& B) {
2     vector<int> intersect;
3     for (int i = 0; i < A.size(); ++i) {
4       if ((i == 0 || A[i] != A[i - 1]) &&
```

```
5           binary_search(B.cbegin(), B.cend(), A[i])) {
6         intersect.emplace_back(A[i]);
7     }
8   }
9   return intersect;
10 }
```

The time complexity is $O(n \log m)$.
We can further improve our run time by choosing the shorter array for the outer 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 the best when the array lengths are similar because we are not exploiting the fact that both arrays are sorted. In this case, iterating in tandem through the elements of each array in increasing order will work best as shown in this code.

```
1  vector<int> intersect_arrs3(const vector<int>& A, const vector<int>& B) {
2    vector<int> intersect;
3    int i = 0, j = 0;
4    while (i < A.size() && j < B.size()) {
5      if (A[i] == B[j] && (i == 0 || A[i] != A[i - 1])) {
6        intersect.emplace_back(A[i]);
7        ++i, ++j;
8      } else if (A[i] < B[j]) {
9        ++i;
10     } else {  // A[i] > B[j].
11       ++j;
12     }
13   }
14   return intersect;
15 }
```

The run time for this algorithm is $O(m + n)$.

Problem 14.2, pg. 72: *Given a set of n events, how would you determine the maximum number of events that take place concurrently?*

Solution 14.2: 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: 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 $b_i \le t \le e_i$).

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.

```
1  struct Interval {
2    int start, finish;
3  };
4
5  struct Endpoint {
6    bool operator<(const Endpoint& e) const {
7      return time != e.time ? time < e.time : (isStart && !e.isStart);
8    }
9
10   int time;
11   bool isStart;
12 };
13
14 int find_max_concurrent_events(const vector<Interval>& A) {
15   // Build the endpoint array.
16   vector<Endpoint> E;
17   for (const Interval& i : A) {
18     E.emplace_back(Endpoint{i.start, true});
19     E.emplace_back(Endpoint{i.finish, false});
20   }
21   // Sort the endpoint array according to the time.
22   sort(E.begin(), E.end());
23
24   // Find the maximum number of events overlapped.
25   int max_count = 0, count = 0;
26   for (const Endpoint& e : E) {
27     if (e.isStart) {
28       max_count = max(++count, max_count);
29     } else {
30       --count;
31     }
32   }
33   return max_count;
34 }
```

Sorting the endpoint array takes $O(n \log n)$ time; iterating through the sorted array takes $O(n)$ time, yielding an $O(n \log n)$ time complexity. The space complexity is $O(1)$.

ϵ-**Variant 14.2.1**: 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 14.3, pg. 73: *Write a function which takes as input an array A of disjoint closed intervals with integer endpoints, sorted by increasing order of left endpoint, and an interval I, and returns the union of I with the intervals in A, expressed as a union of disjoint intervals.*

Solution 14.3: Let $I = [x, y]$. There are two possibilities—A has an interval that has a nonempty intersection with I, or it does not. If it does not contain an interval intersecting I, we simply add I in the appropriate place.

If *A* does contain an interval with a nonempty intersection with *I*, we iterate through *A* until we encounter the first such interval, call it *I'*. As a general fact, given any two intervals [*a*, *b*] and [*a'*, *b'*] that intersect, their union is [min(*a*, *a'*), max(*b*, *b'*)]. If *I'* ∪ *I* = *I'*, there there is nothing to do, we can return *A*. Otherwise, we compute *w* = *I'* ∪ *I*. Now we keep testing subsequent intervals for intersection with *w*. If an interval *J* overlaps with *w* we update *w* to the union of *J* with *w*. As soon as an interval is disjoint from *w*, we add *w* and the remaining intervals to the result and return.

```cpp
struct Interval {
  int left, right;
};

vector<Interval> insert_interval(const vector<Interval>& intervals,
                                 Interval new_interval) {
  size_t i = 0;
  vector<Interval> res;
  // Insert intervals appeared before new_interval.
  while (i < intervals.size() && new_interval.left > intervals[i].right) {
    res.emplace_back(intervals[i++]);
  }

  // Merges intervals that overlap with new_interval.
  while (i < intervals.size() && new_interval.right >= intervals[i].left) {
    new_interval = {min(new_interval.left, intervals[i].left),
                    max(new_interval.right, intervals[i].right)};
    ++i;
  }
  res.emplace_back(new_interval);

  // Insert intervals appearing after new_interval.
  res.insert(res.end(), intervals.begin() + i, intervals.end());
  return res;
}
```

Since the program spends $O(1)$ time per entry, its time complexity is $O(n)$.

Problem 15.1, *pg. 74*: *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 15.1: 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.

```
 1  bool is_BST(const unique_ptr<BinaryTreeNode<int>>& r) {
 2    return is_BST_helper(r,
 3                         numeric_limits<int>::min(),
 4                         numeric_limits<int>::max());
 5  }
 6
 7  bool is_BST_helper(const unique_ptr<BinaryTreeNode<int>>& r,
 8                     int lower, int upper) {
 9    if (!r) {
10      return true;
11    } else if (r->data < lower || r->data > upper) {
12      return false;
13    }
14
15    return is_BST_helper(r->left, lower, r->data) &&
16           is_BST_helper(r->right, r->data, upper);
17  }
```

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.

```
 1  bool is_BST(const unique_ptr<BinaryTreeNode<int>>& root) {
 2    auto* n = root.get();
 3    // Store the value of previous visited node.
 4    int last = numeric_limits<int>::min();
 5    bool res = true;
 6
 7    while (n) {
 8      if (n->left.get()) {
 9        // Find the predecessor of n.
```

```
10          auto* pre = n->left.get();
11          while (pre->right.get() && pre->right.get() != n) {
12            pre = pre->right.get();
13          }
14
15          // Process the successor link.
16          if (pre->right.get()) {  // pre->right == n.
17            // Revert the successor link if predecessor's successor is n.
18            pre->right.release();
19            if (last > n->data) {
20              res = false;
21            }
22            last = n->data;
23            n = n->right.get();
24          } else {  // if predecessor's successor is not n.
25            pre->right.reset(n);
26            n = n->left.get();
27          }
28        } else {
29          if (last > n->data) {
30            res = false;
31          }
32          last = n->data;
33          n = n->right.get();
34        }
35      }
36    return res;
37  }
```

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 depths $d + 1$ 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.

```
1  struct QNode {
2    const unique_ptr<BinaryTreeNode<int>>& node;
3    int lower, upper;
4  };
5
6  bool is_BST(const unique_ptr<BinaryTreeNode<int>>& r) {
7    queue<QNode> q;
8    q.emplace(QNode{r, numeric_limits<int>::min(), numeric_limits<int>::max()});
9
10   while (!q.empty()) {
11     if (q.front().node.get()) {
12       if (q.front().node->data < q.front().lower ||
13           q.front().node->data > q.front().upper) {
14         return false;
15       }
16
17       q.emplace(QNode{q.front().node->left, q.front().lower,
18                       q.front().node->data});
19       q.emplace(QNode{q.front().node->right, q.front().node->data,
20                       q.front().upper});
21     }
22     q.pop();
23   }
24   return true;
25 }
```

Problem 15.2, pg. 75: *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 traversal. If k is absent or no key larger than k is present, return* null. *For example, when applied to the BST in Figure 15.1 on Page 75 you should return 29 if k = 23; if k = 32, you should return* null.

Solution 15.2: A 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.

```
1  BSTNode<int>* find_first_larger_k_with_k_exist(
2      const unique_ptr<BSTNode<int>>& T,
3      int k) {
4    bool found_k = false;
5    BSTNode<int>* curr = T.get(), *first = nullptr;
6
7    while (curr) {
8      if (curr->data == k) {
```

```
9          found_k = true;
10         curr = curr->right.get();
11     } else if (curr->data > k) {
12         first = curr;
13         curr = curr->left.get();
14     } else {   // curr->data < k.
15         curr = curr->right.get();
16     }
17   }
18   return found_k ? first : nullptr;
19 }
```

The time complexity is $O(h)$, where h is the height of the tree. The space complexity is $O(1)$.

Problem 15.3, pg. 75: *How would you build a BST of minimum possible height from a sorted array A?*

Solution 15.3: 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 $A[\lfloor \frac{n}{2} \rfloor + 1 : n - 1]$.

```
1 BSTNode<int>* build_BST_from_sorted_array(const vector<int>& A) {
2   return build_BST_from_sorted_array_helper(A, 0, A.size());
3 }
4
5 // Build BST based on subarray A[start : end - 1].
6 BSTNode<int>* build_BST_from_sorted_array_helper(const vector<int>& A,
7                                                   size_t start, size_t end) {
8   if (start < end) {
9     size_t mid = start + ((end - start) >> 1);
10    return new BSTNode<int>{
11        A[mid], unique_ptr<BSTNode<int>>(
12                    build_BST_from_sorted_array_helper(A, start, mid)),
13        unique_ptr<BSTNode<int>>(
14            build_BST_from_sorted_array_helper(A, mid + 1, end))};
15  }
16  return nullptr;
17 }
```

The time complexity $T(n)$ satisfies the recurrence $T(n) = 2T(n/2) + O(1)$, which solves to $T(n) = O(n)$. Another way of seeing this is that we spend make exactly n calls to the recursive function and spend $O(1)$ in each.

Problem 16.1, pg. 76: *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 16.1: 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 from P1 to P2.

(3.) Recursively transfer the $n - 1$ rings on $P3$ to $P2$, using $P1$.
This is illustrated in Figure 21.3. Code implementing this idea is given below.

(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!
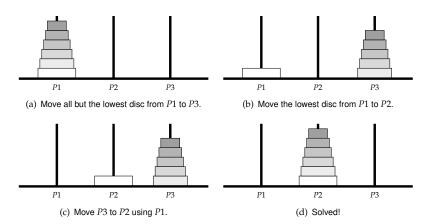
Figure 21.3: A recursive solution to the Towers of Hanoi for $n = 6$.

```cpp
void move_tower_hanoi(int n) {
  array<stack<int>, 3> pegs;
  // Initialize pegs.
  for (int i = n; i >= 1; --i) {
    pegs[0].push(i);
  }

  transfer(n, pegs, 0, 1, 2);
}

void transfer(int n, array<stack<int>, 3>& pegs, int from, int to, 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);
  }
}
```

The number of moves, $T(n)$, satisfies the following recurrence: $T(n) = T(n - 1) + 1 + T(n - 1)$, which solves to $T(n) = 2^n - 1$. One way to see this is to "unwrap" the recurrence k time : $T(n) = 2^l T(n - k) + 2^{k-1} + 2^{k-2} + \cdots + 2 + 1$. Printing a single move takes $O(1)$ time, implying an $O(2^n)$ time complexity. (Note that if we were asked instead to print the smallest number of moves, rather than the moves themselves, we could simply return $2^n - 1$.)

ϵ-**Variant 16.1.1**: Find the minimum number of operations subject to the constraint that each operation must involve $P3$.

ε-**Variant** **16.1**.2 : Find the minimum number of operations subject to the constraint that each transfer must be from *P*1 to *P*2, *P*2 to *P*3, or *P*3 to *P*1.

ε-**Variant** **16.1**.3 : Find the minimum number of operations subject to the constraint that a ring can never be transferred directly from *P*1 to *P*2 (transfers from *P*2 to *P*1 are allowed).

ε-**Variant** **16.1**.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 **16.1**.5 : You have 2*n* disks of *n* different sizes, two of each size. You cannot place a larger disk on a smaller disk, but can place a disk of equal size one on top of the other. Compute the minimum number of moves to transfer the 2*n* disks from *P*1 to *P*2.

Variant **16.1**.6 : You have 2*n* disks which are colored black or white. You cannot place a white disk directly on top of a black disk. Compute the minimum number of moves to complete the transfer to transfer the 2*n* disks from *P*1 to *P*2.

Variant **16.1**.7 : Find the minimum number of operations if you have a fourth peg, *P*4.

Problem **16.2**, *pg.* **76** : *Implement a method that takes as input a set S of n distinct elements, and prints the power set of S. Print the subsets one per line, with elements separated by commas.*

Solution **16.2**: One way to solve this problem is realizing that for a given ordering of the elements of *S*, there exists a one-to-one correspondence between the 2^n bit arrays of length *n* and the set of all subsets of *S*—the 1s in the *n*-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 *n* is less than or equal to the width of an integer on the architecture (or language) we are working on, we can enumerate bit arrays by enumerating integers in $[0, 2^n - 1]$ and examining the indices of bits set in these integers. These indices are determined by first isolating the lowest set bit by computing $y = x\&\sim(x - 1)$, which is described on Page on Page 24 and then getting the index by computing lg *y*.

```
1 void generate_power_set(const vector<int>& S) {
2   for (int i = 0; i < (1 << S.size()); ++i) {
3     int x = i;
4     while (x) {
5       int tar = log2(x & ~(x - 1));
6       cout << S[tar];
```

```
7        if (x &= x - 1) {
8          cout << ',';
9        }
10     }
11     cout << endl;
12   }
13 }
```

Since each set takes $O(n)$ time to print, the time complexity is $O(n2^n)$. In practice, it would likely be faster to iterate through all the bits in x, one at a time.

Alternately, we can use recursion. The most natural recursive algorithm entails recursively computing all subsets U of $S - \{x\}$, i.e., S without the element x (which could be any element), and then adding x to each subset in U to create the set of subsets V. (The base case is when S is empty, in which case we return $\{\{\}\}$.) The subsets in U are distinct from those in V, and the final result is just $U \cup V$, which we can then print. The number of recursive calls for a set of n elements, $T(n)$ satisfies $T(n) = 2T(n-1)$, with $T(0) = 1$, which solves to $T(n) = 2^n$. Since each call takes time $O(n)$, the total time complexity is $O(n2^n)$.

The problem with the approach just described is that it uses $O(n2^n)$ space. The fact that we only need to print the subsets, and not return them, suggests that a more space optimal approach would be to compute the subsets incrementally. Conceptually, the algorithm shown below passes two additional parameters, m and `subset`; the latter indicates which of the first m elements of S must be part of the subsets begin created from the remaining $n - m$ elements. It iteratively prints `subset`, and then prints all subsets of the remaining elements, with and without the $(m + 1)$-th element.

```cpp
1 void generate_power_set(const vector<int>& S) {
2   vector<int> subset;
3   generate_power_set_helper(S, 0, &subset);
4 }
5
6 void generate_power_set_helper(const vector<int>& S, int m,
7                                  vector<int>* subset) {
8   if (!subset->empty()) {
9     // Print the subset.
10     copy(subset->cbegin(), subset->cend() - 1,
11          ostream_iterator<int>(cout, ","));
12     cout << subset->back();
13   }
14   cout << endl;
15
16   for (int i = m; i < S.size(); ++i) {
17     subset->emplace_back(S[i]);
18     generate_power_set_helper(S, i + 1, subset);
19     subset->pop_back();
20   }
21 }
```

The number of recursive calls, $T(n)$ satisfies the recurrence $T(n) = T(n-1) + T(n-2) + \cdots + T(1) + T(0)$, which solves to $T(n) = O(2^n)$. Since we spend $O(n)$ time within a

call, the time complexity is $O(n2^n)$. The space complexity is $O(n)$ which comes from the maximum stack depth as well as the maximum size of a subset.

Variant 16.2.1: Print all subsets of size k of $\{1, 2, 3, \ldots, n\}$.

Problem 16.3, pg. 77: *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 16.3: 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.

```
1  bool solve_Sudoku(vector<vector<int>>* A) {
2    if (!is_valid_Sudoku(*A)) {
3      cout << "Initial configuration violates constraints." << endl;
4      return false;
5    }
6
7    if (solve_Sudoku_helper(0, 0, A)) {
8      for (int i = 0; i < A->size(); ++i) {
9        copy((*A)[i].begin(), (*A)[i].end(), ostream_iterator<int>(cout, " "));
10       cout << endl;
11     }
12     return true;
13   } else {
14     cout << "No solution exists." << endl;
15     return false;
16   }
17 }
18
19 bool solve_Sudoku_helper(int i, int j, vector<vector<int>>* A) {
20   if (i == A->size()) {
21     i = 0;  // starts a new row.
22     if (++j == (*A)[i].size()) {
23       return true;  // Entire matrix has been filled without conflict.
24     }
25   }
26
27   // Skips nonempty entries.
28   if ((*A)[i][j] != 0) {
29     return solve_Sudoku_helper(i + 1, j, A);
30   }
31
32   for (int val = 1; val <= A->size(); ++val) {
33     // Note: practically, it's substantially quicker to check if entry val
34     // conflicts with any of the constraints if we add it at (i,j) before
35     // adding it, rather than adding it and then calling is_valid_Sudoku.
```

```
36      // The reason is that we know we are starting with a valid configuration,
37      // and the only entry which can cause a problem is entryval at (i,j).
38      if (valid_to_add(*A, i, j, val)) {
39        (*A)[i][j] = val;
40        if (solve_Sudoku_helper(i + 1, j, A)) {
41          return true;
42        }
43      }
44    }
45
46    (*A)[i][j] = 0;  // undo assignment.
47    return false;
48 }
49
50 bool valid_to_add(const vector<vector<int>>& A, int i, int j, int val) {
51    // Check row constraints.
52    for (int k = 0; k < A.size(); ++k) {
53      if (val == A[k][j]) {
54        return false;
55      }
56    }
57
58    // Check column constraints.
59    for (int k = 0; k < A.size(); ++k) {
60      if (val == A[i][k]) {
61        return false;
62      }
63    }
64
65    // Check region constraints.
66    int region_size = sqrt(A.size());
67    int I = i / region_size, J = j / region_size;
68    for (int a = 0; a < region_size; ++a) {
69      for (int b = 0; b < region_size; ++b) {
70        if (val == A[region_size * I + a][region_size * J + b]) {
71          return false;
72        }
73      }
74    }
75    return true;
76 }
```

Because the program is specialized to 9×9 grids, it does not make sense to speak of its time complexity, since there is no notion of scaling with a size parameter. However, since the problem of solving Sudoku generalized to $n \times n$ grids is NP-complete, it should not be difficult to prove that the generalization of this algorithm to $n \times n$ grids has exponential time complexity.

Variant 16.3.1: Compute a placement of eight queens on an 8×8 chessboard in which no two queens attack each other.

Variant 16.3.2: 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 16.3.3: Compute the smallest number of queens that can be placed to attack each uncovered square.

Problem 17.1, pg. 81: *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 17.1: 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 $t \le s$ 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.

```
1  int count_combinations(int k, const vector<int>& score_ways) {
2    vector<int> combinations(k + 1, 0);
3    combinations[0] = 1;  // one way to reach 0.
4    for (const int& score : score_ways) {
5      for (int j = score; j <= k; ++j) {
6        combinations[j] += combinations[j - score];
7      }
8    }
9    return combinations[k];
10 }
```

It should be clear that the time complexity is $O(sn)$ (two loops, one to s, the other to n) and space complexity is $O(s)$ (the combinations array).

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 $t < s$.

```
1  int count_permutations(int k, const vector<int>& score_ways) {
2    vector<int> permutations(k + 1, 0);
3    permutations[0] = 1;  // one way to reach 0.
4    for (int i = 0; i <= k; ++i) {
```

```
5      for (const int& score : score_ways) {
6        if (i >= score) {
7          permutations[i] += permutations[i - score];
8        }
9      }
10   }
11   return permutations[k];
12 }
```

The time and space complexities are the same as those for computing the number of combinations, i.e., $O(sn)$ and $O(s)$, respectively.

Variant 17.1.1: 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 17.1.2: 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 17.2, pg. 81: *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 17.2: 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 17.2 on Page 81 is shown in Figure 21.4 on the next page. To fill in the i-th row we do not need values from rows before $i - 1$. Consequently, we do not need an $n \times m$ 2D array. Instead, we use two rows of storage, alternating between By symmetry, the number of ways to get to (i, j) is the same as to get to (j, i), so we use the smaller of m and n to be the number of columns (which is the number of entries in a row). This leads to the following algorithm.

```
1 int number_of_ways(int n, int m) {
2   if (n < m) {
3     swap(n, m);
4   }
5   vector<vector<int>> A(2, vector<int>(m, 1));
6   for (int i = 1; i < n; ++i) {
7     for (int j = 0; j < m; ++j) {
8       A[i & 1][j] = (i < 1 ? 0 : A[(i - 1) & 1][j]) +
9                     (j < 1 ? 0 : A[i & 1][j - 1]);
10    }
```

```
11    }
12    return A[(n - 1) & 1][m - 1];
13 }
```

The time complexity is $O(mn)$, and the space complexity is $O(\min(m, n))$.

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 21.4: The number of ways to get from $(0, 0)$ to (i, j) for $0 \le i, j \le 4$.

Another way of deriving the same answer is to use 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.

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).

```
1  // Given the dimensions of A, n and m, and B, return the number of ways
2  // from A[0][0] to A[n - 1][m - 1] considering obstacles.
3  int number_of_ways_with_obstacles(int n, int m,
4                                    const vector<deque<bool>> &B) {
5    vector<vector<int>> A(n, vector<int>(m, 0));
6    if (B[0][0]) {  // no way to start from (0, 0) if B[0][0] == true.
7      return 0;
8    } else {
9      A[0][0] = 1;
10   }
11   for (int i = 0; i < n; ++i) {
12     for (int j = 0; j < m; ++j) {
13       if (B[i][j] == 0) {
14         A[i][j] += (i < 1 ? 0 : A[i - 1][j]) + (j < 1 ? 0 : A[i][j - 1]);
15       }
16     }
17   }
18   return A.back().back();
19 }
```

The time complexity is $O(nm)$.

Variant 17.2.1: 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 $D[i] \le D[i+1], 0 \le i < |D|$. 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 17.2.2: 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 17.3, pg. 82: *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 17.3: 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:

$$V[i, w] = \begin{cases} \max\left(V[i - 1, w], V[i - 1, w - w_i] + v_i\right), & \text{if } w_i \leq w; \\ V[i - 1, w], & \text{otherwise.} \end{cases}$$

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. 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.

```
1  struct Item {
2    int weight, value;
3  };
4
5  int knapsack(int w, const vector<Item>& items) {
6    vector<int> V(w + 1, 0);
7    for (int i = 0; i < items.size(); ++i) {
8      for (int j = w; j >= items[i].weight; --j) {
9        V[j] = max(V[j], V[j - items[i].weight] + items[i].value);
10     }
11   }
12   return V[w];
13 }
```

Variant 17.3.1: Solve the knapsack problem when the thief can take a fractional amount of an item.

Problem 17.4, pg. 82: *Given a dictionary, i.e., a set of strings, 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 17.4: 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. Then $T[k]$ holds iff one of the following two conditions is true:

- Substring $s(0, k)$ is a valid word.
- There exists a $j \in [0, k - 1]$ such that $T[j]$ is true and $s(j + 1, k)$ 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.

```
1  vector<string> word_breaking(const string& s,
2                               const unordered_set<string>& dict) {
3    // T[i] is the length of the last string in the decomposition of s(0, i).
4    vector<int> T(s.size(), 0);
5    for (int i = 0; i < s.size(); ++i) {
6      // Set T[i] if s(0, i) is a valid word.
7      if (dict.find(s.substr(0, i + 1)) != dict.cend()) {
8        T[i] = i + 1;
9      }
10
11     // Set T[i] if T[j] != 0 and s(j + 1, i) is a valid word.
12     for (int j = 0; j < i && T[i] == 0; ++j) {
13       if (T[j] != 0 && dict.find(s.substr(j + 1, i - j)) != dict.cend()) {
14         T[i] = i - j;
15       }
16     }
17   }
18
19   vector<string> ret;
20   // s can be assembled by valid words.
21   if (T.back()) {
22     int idx = s.size() - 1;
23     while (idx >= 0) {
24       ret.emplace_back(s.substr(idx - T[idx] + 1, T[idx]));
25       idx -= T[idx];
26     }
27     reverse(ret.begin(), ret.end());
28   }
29   return ret;
30 }
```

For each k we check for each $j < k$ whether $s(j + 1, k)$ is a valid word, and each such check requires $O(k - j)$ time. This implies the time complexity is $O(n^3)$. Let W be the length of the longest valid word. By changing j to range from $k - W$ to $k - 1$, we can improve the time complexity to $O(n^2 W)$. Note that we are assuming the dictionary is specified as a hash table.

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...".

Variant 17.4.1: Devise an $O(nW)$ algorithm for word breaking.

Problem 17.5, pg. 82: *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* $A[i_j] \le A[i_{j+1}]$ *for any* $j \in [0, k - 2]$.

Solution 17.5: We present two solutions, an $O(n^2)$, and an $O(n \log n)$ one.

We first describe the $O(n^2)$ solution, which is a straightforward application of dynamic programming. 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:

$$s_i = \max_{j \in [0, i-1]} \left(\begin{array}{ll} s_j + 1, & \text{if } A[j] \le A[i]; \\ 1, & \text{otherwise.} \end{array} \right)$$

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:

```cpp
vector<int> longest_nondecreasing_subsequence(const vector<int>& 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<int> 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. It is not based on dynamic programming—it is a greedy algorithm and uses binary search. 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 $j' \leq j$.

Now suppose there does not exist j such that $M_{i,j} \leq A[i+1]$. 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.

```
1  int longest_nondecreasing_subsequence(const vector<int>& A) {
2    vector<int> tail_values;
3    for (const int& a : A) {
4      auto it = upper_bound(tail_values.begin(), tail_values.end(), a);
5      if (it == tail_values.end()) {
6        tail_values.emplace_back(a);
7      } else {
8        *it = a;
9      }
10   }
11   return tail_values.size();
12 }
```

ϵ-**Variant 17.5.1**: 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 17.5.2**: 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 17.5.3**: 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 $1 \leq i \leq n - 2$. 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 17.5.4**: 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 $0 \leq i < k$ and $a_i > a_{i+1}$, for $k \leq i < n - 1$. 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 17.5.5**: 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 18.1, pg. 84: *Given a set of symbols with corresponding frequencies, find a code book that has the smallest average code length.*

Solution 18.1: 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.

```
1  struct Symbol {
2    char c;
3    double prob;
4    string code;
5  };
6
7  struct BinaryTreeNode {
8    double prob;
9    Symbol* s;
10   shared_ptr<BinaryTreeNode> left, right;
11 };
12
13 struct Compare {
14   bool operator()(const shared_ptr<BinaryTreeNode>& lhs,
15                   const shared_ptr<BinaryTreeNode>& rhs) {
16     return lhs->prob > rhs->prob;
17   }
18 };
19
20 void Huffman_encoding(vector<Symbol>* symbols) {
21   // Initially assign each symbol into min->heap.
22   priority_queue<shared_ptr<BinaryTreeNode>,
23                  vector<shared_ptr<BinaryTreeNode>>,
24                  Compare> min_heap;
25   for (auto& s : *symbols) {
26     min_heap.emplace(new BinaryTreeNode{s.prob, &s, nullptr, nullptr});
27   }
28
29   // Keep combining two nodes until there is one node left.
30   while (min_heap.size() > 1) {
31     shared_ptr<BinaryTreeNode> l = min_heap.top();
32     min_heap.pop();
33     shared_ptr<BinaryTreeNode> r = min_heap.top();
34     min_heap.pop();
35     min_heap.emplace(new BinaryTreeNode{l->prob + r->prob, nullptr, l, r});
```

```
36   }
37
38   // Traverse the binary tree and assign code.
39   assign_huffman_code(min_heap.top(), string());
40 }
41
42 // Traverse tree and assign code.
43 void assign_huffman_code(const shared_ptr<BinaryTreeNode>& r,
44                          const string& s) {
45   if (r) {
46     // This node (i.e.,leaf) contains symbol.
47     if (r->s) {
48       r->s->code = s;
49     } else {  // non-leaf node.
50       assign_huffman_code(r->left, s + '0');
51       assign_huffman_code(r->right, s + '1');
52     }
53   }
54 }
```

Applying this algorithm to the frequencies for English characters presented in Table 18.1 on Page 85 yields the Huffman tree in Figure 21.5. 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.
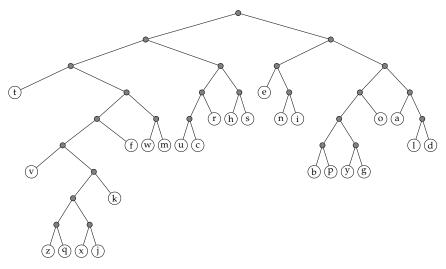
Figure 21.5: A Huffman tree for the English characters, assuming the frequencies given in Table 18.1 on Page 85.

The codebook is explicitly given in Table 21.1 on the following page. The average code length for this coding is 4.205. In contrast, the trivial coding takes ⌈log 26⌉ = 5 bits for each character.

It is exceedingly unlikely that you would be asked for a rigorous proof of optimality would be asked in an interview setting. The reasoning behind the Huffman

Table 21.1: Huffman codes for English characters, assuming the frequencies given in Table 18.1 on Page 85.

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		

algorithm yielding the minimum average code length is based on a induction on the number of symbols. The induction step itself makes use of proof by contradiction, with the two leaves in the Huffman tree corresponding to the rarest symbols playing a central role.

Problem 18.2, pg. 86: *Design an algorithm that takes as input an array A and a number t, and determines if A 3-creates t.*

Solution 18.2: First, we consider the problem of computing a pair of entries which sum to K. Assume A is sorted. We 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.

Now we describe a solution to the problem of finding three entries which sum to t. We 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 described in the previous paragraph n times (one for each entry), which is $O(n^2)$ overall. The code for this approach is shown below.

```
1  bool has_3_sum(vector<int> A, int t) {
2    sort(A.begin(), A.end());
3
4    for (int a : A) {
5      // Find if the sum of two numbers in A equals to t - a.
6      if (has_2_sum(A, t - a)) {
7        return true;
8      }
9    }
10   return false;
11 }
```

```
12
13  bool has_2_sum(const vector<int>& A, int t) {
14    int j = 0, k = A.size() - 1;
15
16    while (j <= k) {
17      if (A[j] + A[k] == t) {
18        return true;
19      } else if (A[j] + A[k] < t) {
20        ++j;
21      } else {  // A[j] + A[k] > t.
22        --k;
23      }
24    }
25    return false;
26  }
```

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 18.2.1**: Solve the same problem when the three elements must be distinct. For example, if $A = \langle 5, 2, 3, 4, 3 \rangle$ 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 18.2.2: Solve the same problem when k is an additional input.

Variant 18.2.3: Write a function that takes as input an array of integers A and an integer T, and returns a 3-tuple $(A[p], A[q], A[r])$ where p, q, r are all distinct, minimizing $|T - (A[p] + A[q] + A[r])|$, and $A[p] \leq A[r] \leq A[s]$.

Problem 18.3, pg. 86: *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 18.3: 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.

We now describe a relatively simple $O(n)$ time algorithm to compute the largest rectangle. We iterate forward through A, keeping a subset of indices seen so far in a

stack S. Specifically, suppose we have completed iterating up to and including index $i - 1$.

If $A[i]$ is greater than or equal to the element corresponding to the index at the top of the stack, we push i on to S and continue. Otherwise, we pop S till $A[i]$ is greater than or equal to the element corresponding to the top of S. Note that this rule ensures that throughout the computation the array elements corresponding to indices in the stack are in nondecreasing order from the bottom to top of the stack. After all indices j in the stack for which $A[j] > A[i]$ have been popped, we push i onto the stack and continue.

Now we describe the computations performed for each pop.

For each i, for every index j that is popped, because $A[i] < A[j]$, no rectangle of height $A[j]$ that includes j can extend past $i - 1$. Furthermore, there can be no k, $j < k < i$ such that $A[k] < A[j]$, or we would have popped j earlier. Therefore, the farthest to the right a rectangle of height $A[j]$ that includes j can reach is exactly $i - 1$.

Now we consider how far to the left a rectangle of height $A[j]$ that includes j can reach. There are three possibilities.

  - There is no entry below j in the stack. The farthest to the left a rectangle of height $A[j]$ that begins at j can extend to is 0.
  - There is an entry j' below j in the stack, and $A[j'] < A[j]$. Then the farthest to the left a rectangle of height $A[j]$ that begins at j can extend to is $j' + 1$. (There cannot be k, $j' < k < j$ such that $A[k] < A[j']$, because we would have popped j' when processing k.) The rectangle from $j' + 1$ to $i - 1$ is the widest any rectangle including j whose height is $A[j]$—it is blocked on each side from going further.
  - There is an entry j' below j in the stack, and $A[j'] = A[j]$. As before, the rectangle from $j' + 1$ to $i - 1$ is entirely under the skyline. It is *not* the widest rectangle including j whose height is $A[j]$, since it is not blocked by j'. However, after j is processed j' is guaranteed to be processed (since $A[j'] = A[j] > A[i]$), and we will eventually identify the widest rectangle including j whose height is $A[j]$ when processing i.

The computation described above is almost complete. There is one possibility that it does not consider, namely that the stack is not empty when we reach the end. The computation is analogous to the iteration over i. Let j be the index at the top of the stack. Suppose there is an entry j' below j in the stack. Then the widest rectangle including j with height $A[j]$ extends up to and including $n - 1$ on the right. On the left, it extends to at least $j' + 1$, and further, if $A[j'] = A[j]$. If there is no entry below j, then the widest rectangle j with height $A[j]$ extends up to and including 0 on the left.

Through the entire computation, we compare the area of the rectangles computed to the maximum recorded so far and conditionally update that maximum. Since a largest rectangle under the skyline corresponds to some index, and for each index we identify the area of the largest rectangle including that index, with corresponding height, the algorithm described above correctly computes the area of a largest rectangle under the skyline.

The processing performed during the iteration over $i = 0$ to $n - 1$ is very similar to the processing performed when $i = n$. It can be unified into a single loop using appropriate loop conditions, as show below.

```cpp
int calculate_largest_rectangle(const vector<int>& A) {
  stack<int> s;
  int max_area = 0;
  for (int i = 0; i <= A.size(); ++i) {
    while (!s.empty() && (i == A.size() || A[i] < A[s.top()])) {
      int height = A[s.top()];
      s.pop();
      max_area = max(max_area, height * (s.empty() ? i : i - s.top() - 1));
    }
    s.emplace(i);
  }
  return max_area;
}
```

The time complexity is $O(n)$. In the forward pass, the time spent for each i is proportional to the number of pushes and pops performed in that iteration. Although for some i we may perform multiple pops, in total we perform exactly n pushes and at most n pops. This is because in the forward iteration, each i is added once to the stack and cannot be popped more than once. The time complexity of the second stage is also $O(n)$ since there are at most n elements in the stack, and the time to process each one is $O(1)$, which yields the overall time complexity of $O(n)$. The space complexity is $O(n)$, which is the largest the stack can grow to, e.g., if A is sorted in ascending order.

ϵ-**Variant 18.3.1**: Find the largest square under the skyline.

Problem 19.1, pg. 90: *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 19.1: 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.

```
1  struct Coordinate {
2    bool operator==(const Coordinate& that) const {
3      return x == that.x && y == that.y;
4    }
5
6    int x, y;
7  };
8
9  vector<Coordinate> search_maze(vector<vector<int>> maze,
10                                 const Coordinate& s,
11                                 const Coordinate& e) {
12   vector<Coordinate> path;
13   maze[s.x][s.y] = 1;
14   path.emplace_back(s);
15   if (!search_maze_helper(s, e, &maze, &path)) {
16     path.pop_back();
17   }
18   return path;  // empty path means no path between s and e.
19 }
20
21 // Perform DFS to find a feasible path.
22 bool search_maze_helper(const Coordinate& cur,
23                         const Coordinate& e,
24                         vector<vector<int>>* maze,
25                         vector<Coordinate>* path) {
26   if (cur == e) {
27     return true;
28   }
29
30   const array<array<int, 2>, 4> shift = {
31       {{{0, 1}}, {{0, -1}}, {{1, 0}}, {{-1, 0}}}};
32
33   for (const auto& s : shift) {
34     Coordinate next{cur.x + s[0], cur.y + s[1]};
35     if (is_feasible(next, *maze)) {
36       (*maze)[next.x][next.y] = 1;
37       path->emplace_back(next);
38       if (search_maze_helper(next, e, maze, path)) {
39         return true;
40       }
41       path->pop_back();
42     }
43   }
44   return false;
45 }
46
47 // Check cur is within maze and is a white pixel.
48 bool is_feasible(const Coordinate& cur, const vector<vector<int>>& maze) {
49   return cur.x >= 0 && cur.x < maze.size() && cur.y >= 0 &&
50          cur.y < maze[cur.x].size() && maze[cur.x][cur.y] == 0;
51 }
```

The time complexity is the same as that for DFS, namely $O(|V| + |E|)$.

Problem 19.2, pg. 90: *Implement a routine that takes a n × m Boolean array A together with an entry (x, y) and flips the color of the region associated with (x, y). See Figure 19.6 on Page 91 for an example of flipping.*

Solution 19.2: Conceptually, this problem is very similar to that of exploring an undirected graph. Entries can be viewed as vertices, and neighboring vertices are connected by edges.

For the current problem, we are searching for all vertices whose color is the same as that of (x, y) that are reachable from (x, y). Breadth-first search is natural when starting with a set of vertices. Specifically, we can use a queue to store such vertices. The queue is initialized to (x, y). The queue is popped iteratively. Call the popped point p. First, we record p's initial color, and then flip its color. Next we examine p neighbors. Any neighbor which is the same color as p's initial color is added to q. The computation ends when q is empty. Correctness follows from the fact that any point that is added to the queue is reachable from (x, y) via a path consisting of points of the same color, and all points reachable from (x, y) via points of the same color will eventually be added to the queue.

```cpp
void flip_color(int x, int y, vector<deque<bool>> *A) {
  const array<array<int, 2>, 4> dir = {{{{0, 1}}, {{0, -1}},
                                        {{1, 0}}, {{-1, 0}}}};
  const bool color = (*A)[x][y];

  queue<pair<int, int>> q;
  (*A)[x][y] = !(*A)[x][y];  // flips.
  q.emplace(x, y);
  while (!q.empty()) {
    auto curr(q.front());
    for (const auto& d : dir) {
      const 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) {
        // Flips the color.
        (*A)[next.first][next.second] = !(*A)[next.first][next.second];
        q.emplace(next);
      }
    }
    q.pop();
  }
}
```

The time complexity is the same as that of BFS, i.e., $O(mn)$. The space complexity is a little better than the worst case for BFS, since there are at most $O(m + n)$ vertices that are at the same distance from a given entry.

We also provide a recursive solution which is in the spirit of DFS. It does not need a queue but implicitly uses a stack, namely the function call stack.

```cpp
void flip_color(int x, int y, vector<deque<bool>> *A) {
```

```
2    const array<array<int, 2>, 4> dir = {{{{0, 1}}, {{0, -1}},
3                                          {{1, 0}}, {{-1, 0}}}};
4    const bool color = (*A)[x][y];
5
6    (*A)[x][y] = !(*A)[x][y];  // flips.
7    for (const auto& d : dir) {
8      const int nx = x + d[0], ny = y + d[1];
9      if (nx >= 0 && nx < A->size() && ny >= 0 && ny < (*A)[nx].size() &&
10         (*A)[nx][ny] == color) {
11       flip_color(nx, ny, A);
12     }
13   }
14 }
```

The time complexity is the same as that of DFS.

Both the algorithms given above differ slightly from traditional BFS and DFS algorithms. The reason is that we have a color field already available, and hence do not need the auxiliary color field traditionally associated with vertices BFS and DFS. Furthermore, since we are simply determining reachability, we only need two colors, whereas BFS and DFS traditionally use three colors to track state. (The use of an additional color makes it possible, for example, to answer questions about cycles in directed graphs, but that is not relevant here.)

ϵ-**Variant 19.2.1**: Design an algorithm for computing the black region that contains the most points.

ϵ-**Variant 19.2.2**: 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 19.3, pg. 90: *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 19.3: Define the undirected graph $G = (D, E)$ by $(u, v) \in E$ iff $n_u = n_v$, where n_u and n_s are the lengths of u and v, respectively, 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.

```
1  // Use BFS to find the least steps of transformation.
2  int transform_string(unordered_set<string> D,
```

```
3                         const string& s,
4                         const string& t) {
5    queue<pair<string, int>> q;
6    D.erase(s);  // mark s as visited by erasing it in D.
7    q.emplace(s, 0);
8
9    while (!q.empty()) {
10     pair<string, int> f(q.front());
11     // Return if we find a match.
12     if (f.first == t) {
13       return f.second;  // number of steps reaches t.
14     }
15
16     // Try all possible transformations of f.first.
17     string str = f.first;
18     for (int i = 0; i < str.size(); ++i) {
19       for (int j = 0; j < 26; ++j) {  // iterates through 'a' ~ 'z'.
20         str[i] = 'a' + j;               // change the (i + 1)-th char of str.
21         auto it(D.find(str));
22         if (it != D.end()) {
23           D.erase(it);  // mark str as visited by erasing it.
24           q.emplace(str, f.second + 1);
25         }
26       }
27       str[i] = f.first[i];  // revert the change of str.
28     }
29     q.pop();
30   }
31
32   return -1;  // cannot find a possible transformations.
33 }
```

The number of vertices is the number d of words in the dictionary. The number of edges is, in the worst case, $O(d^2)$. The time complexity is that of BFS, namely $O(d + d^2) = O(d^2)$. (If the string length n is less than d then the number of edges drops to $O(n)$, implying an $O(nd)$ bound.)

Problem 19.4, *pg.* 92: *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 19.4: 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.

Problem 20.1, *pg. 94*: *Write Java code in which the two threads, running concurrently, print the numbers from 1 to 100 in order.*

Solution 20.1: A brute-force solution is to use a lock which is repeatedly captured by the threads. A single variable, protected by the lock, indicates who went last. The drawback of this approach is that it employs the busy waiting antipattern: processor time that could be used to execute a different task is instead wasted on useless activity.

Below we present a solution based on the same idea, but one that avoids busy locking by using wait() and notify() primitives.

```java
static class OddEvenMonitor {
    public static final boolean ODD_TURN = true;
    public static final boolean EVEN_TURN = false;
    private boolean turn = ODD_TURN;

    public synchronized void waitTurn(boolean oldTurn) {
        while (turn != oldTurn) {
            try {
                wait();
            } catch(Exception e) {
            }
        }
    }

    public synchronized void toggleTurn() {
        turn ^= true;
        notify();
    }
}

static class OddThread extends Thread {
    private final OddEvenMonitor monitor;

    public OddThread(OddEvenMonitor monitor) {
        this.monitor = monitor;
    }
    @Override
    public void run() {
        for (int i = 1; i <= 100; i+=2) {
            monitor.waitTurn(OddEvenMonitor.ODD_TURN);
            System.out.println(i);
            monitor.toggleTurn();
        }
    }
}
```

```
36
37    static class EvenThread extends Thread {
38        private final OddEvenMonitor monitor;
39
40        public EvenThread(OddEvenMonitor monitor) {
41            this.monitor = monitor;
42        }
43        @Override
44        public void run() {
45            for (int i = 2; i <= 100; i+=2) {
46                monitor.waitTurn(OddEvenMonitor.EVEN_TURN);
47                System.out.println(i);
48                monitor.toggleTurn();
49            }
50        }
51    }
```

Problem 20.2, pg. 94: *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 20.2: 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 20.3, pg. 94: *Implement a synchronization mechanism for the first readers-writers problem.*

Solution 20.3: We want to keep track of whether the string is being read from, as well as whether the string is being written to. Additionally, if the string is being read

from, we want to know the number of concurrent readers. 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.

```
1  // LR and LW are static members of type Object in the RW class.
2  // They serve as read and write locks. The static integer
3  // variable readCount in RW tracks the number of readers.
4  class Reader extends Thread {
5    public void run() {
6      while (true) {
7        synchronized (RW.LR) {
8          RW.readCount++;
9        }
10       System.out.println(RW.data);
11       synchronized (RW.LR) {
12         RW.readCount--;
13         RW.LR.notify();
14       }
15       Task.doSomeThingElse();
16     }
17   }
18 }
19
20 class Writer extends Thread {
21   public void run() {
22     while (true) {
23       synchronized (RW.LW) {
24         boolean done = false;
25         while (!done) {
26           synchronized (RW.LR) {
27             if (RW.readCount == 0) {
28               RW.data = new Date().toString();
29               done = true;
30             } else {
31               // use wait/notify to avoid busy waiting
32               try {
33                 // protect against spurious notify, see
34                 // stackoverflow.com do-spurious-wakeups-actually-happen
35                 while ( RW.readCount != 0 ) {
36                   RW.LR.wait();
37                 }
38               } catch (InterruptedException e) {
39                 System.out.println("InterruptedException in Writer wait");
40               }
41             }
42           }
43         }
44       }
```

```
45          Task.doSomeThingElse ();
46       }
47    }
48 }
```

Problem 21.1, pg. 96: *Design a system that can compute the ranks of ten billion web pages in a reasonable amount of time.*

Solution 21.1: 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 21.2, pg. 97: *Design a system that will help its users find mileage runs.*

Solution **21.2**: There are two distinct aspects to the design. The first is the user-facing portion of the system. The second is the server backend that gets flight-price-distance information and combines it with user input to generate the alerts.

We begin with the user-facing portion. For simplicity, we illustrate it with a web-app, with the realization that the web-app could also be written as a desktop or mobile app. The web-app has the following components: a login page, a manage alerts page, a create an alert page, and a results page. For such a system we would like defer to a single-sign-on login service such as that provided by Google or Facebook. The management page would present login information, a list of alerts, and the ability to create an alert.

One reasonable formulation of an alert is that it is an origin city, a target cpm, and optionally, a date or range of travel dates. The results page would show flights satisfying the constraints. Note that other formulations are also possible, such as how frequently to check for flights, a set of destinations, a set of origins, etc.

The classical approach to implement the web-app front end is through dynamically generated HTML on the server, e.g., through Java Server Pages. It can be made more visually appealing and intuitive by making appropriate use of cascaded style sheets, which are used for fonts, colors, and placements. The UI can be made more efficient through the use of Javascript to autocomplete common fields, and make attractive date pickers.

Modern practice is to eschew server-side HTML generation, and instead have a single-page application, in which Javascript reads and writes JavaScript Object Notation (JSON) objects to the server, and incrementally updates the single-page based. The AngularJS framework supports this approach.

The web-app backend server has four components: gathering flight data, matching user-generated alerts to this data, persisting data and alerts, and generating the responses to browser initiated requests.

Flight data can be gathered via "scraping" or by subscribing to a flight data service. Scraping refers to extraction of data from a website. It can be quite involved—some of the issues are parsing the results from the website, filling in form data, and running the Javascript that often populates the actual results on a page. Selenium is a Java library that can programmatically interface to the Firefox browser, and is appropriate for scraping sites that are rich in Javascript. Most flight data services are paid. ITA software provides a very widely used paid aggregated flight data feed service. The popular Kayak site provides an Extensible Markup Language (XML) feed of recently discovered fares, which can be a good free alternative. Flight data does not include the distance between airports, but there are websites which return the distance between airport codes which can be used to generate the cpm for a flight.

There are a number of common web application frameworks—essentially libraries that handle many common tasks—that can be used to generate the server. Java and Python are very commonly used for writing the backend for web applications.

Persistence of data can be implemented through a database. Most web application frameworks provide support for automating the process of reading and writing objects from and to a database. Finally, web application frameworks can route

incoming HTTP requests to appropriate code—this is through a configuration file matching URLs to methods. The framework provides convenience methods for accessing HTTP fields and writing results. Frameworks also provide HTTP templating mechanisms, wherein developers intersperse HTML with snippets of code that dynamically add content to the HTML.

Web application frameworks typically implement cron functionality, wherein specified functions are executed at a regular interval. This can be used to periodically scrape data and check if the condition of an alert is matched by the data.

Finally, the web app can be deployed via a platform-as-a-service such as Amazon Web Services, or built on an application-as-a-service such as Google AppEngine.

Part V

Notation and Index

Notation

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
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
$(d_{k-1} \ldots d_0)_r$	radix-r representation of a number, e.g., $(1011)_2$
$\log_b x$	logarithm of x to the base b
$\lg x$	logarithm of x to the base 2
$|S|$	cardinality of set S
$S \setminus T$	set difference, i.e., $S \cap T'$, sometimes written as $S - T$
$|x|$	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)	open interval: $\{x \mid l < x < r\}$
$[l, r)$	$\{x \mid l \le x < r\}$
$(l, r]$	$\{x \mid l < x \le r\}$
$\{a, b, \dots\}$	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 2D array A
$A[i_1 : i_2][j_1 : j_2]$	2D subarray of 2D 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
$x \bmod y$	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 $\{\dots, -2, -1, 0, 1, 2, 3, \dots\}$
$\mathcal{Z}^+$	the set of nonnegative integers $\{0, 1, 2, 3, \dots\}$
$\mathcal{Z}_n$	the set $\{0, 1, 2, 3, \dots, n - 1\}$
$\mathcal{R}$	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

Index of Terms