

# Elements of Programming Interviews

Adnan Aziz

Amit Prakash

Tsung-Hsien Lee

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 at [Amazon.com](http://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.

**Amit Prakash** is a founder of Scaligent, 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.

**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.

## **Elements of Programming Interviews: 300 Questions and Solutions**

by Adnan Aziz, Amit Prakash, and Tsung-Hsien Lee

Copyright © 2013 Adnan Aziz, Amit Prakash, and Tsung-Hsien Lee. All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the authors.

The views and opinions expressed in this work are those of the authors and do not necessarily reflect the official policy or position of their employers.

We typeset this book using  $\text{\LaTeX}$  and the Memoir class. We used TikZ to draw figures. Allan Ytac created the cover, based on a design brief we provided.

The companion website for the book includes contact information and a list of known errors for each version of the book. If you come across an error or an improvement, please let us know.

Version 1.3.3

Website: <http://ElementsOfProgrammingInterviews.com>

Distributed under the Attribution-NonCommercial-NoDerivs 3.0 License



---

# Table of Contents

<b>I</b>	<b>The Interview</b>	<b>6</b>
	1 Getting Ready · 7	
	2 Strategies For A Great Interview · 11	
	3 Conducting An Interview · 18	
	4 Problem Solving Patterns · 22	
<b>II</b>	<b>Problems</b>	<b>45</b>
	5 Primitive Types · 46	
	6 Arrays and Strings · 49	
	7 Linked Lists · 53	
	8 Stacks and Queues · 56	
	9 Binary Trees · 59	
	10 Heaps · 63	
	11 Searching · 66	
	12 Hash Tables · 71	
	13 Sorting · 73	
	14 Binary Search Trees · 76	
	15 Meta-algorithms · 78	

	16 Graphs · 85
	17 Intractability · 91
	18 Parallel Computing · 95
	19 Design Problems · 98
	20 Probability · 100
	21 Discrete Mathematics · 103
<b>III</b>	<b>Hints 106</b>
	22 Hints · 107
<b>IV</b>	<b>Solutions 110</b>
<b>V</b>	<b>Notation and Index 200</b>
	Index of Terms · 203

# Introduction

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

— N. MACHIAVELLI, 1513

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

## *An interview problem*

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

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

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

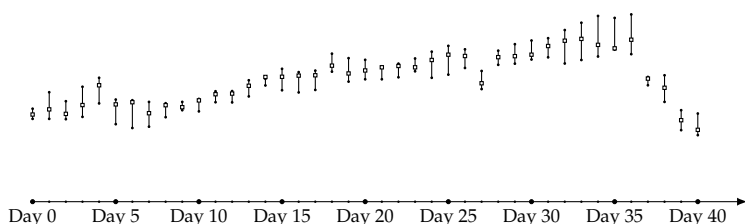


Figure 1: Share price as a function of time.

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 118.

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

The notation, specifically the symbols we use for describing algorithms, e.g.,  $|S|, A[i : j]$ , is summarized starting on Page 201; we strongly recommend that you review it. Terms, e.g., BFS and dequeue, are indexed starting on Page 203.

### ***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 machinery, 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 111. A guide to reading C++ programs for Java developers is given on Page 111. Source code, which includes randomized and directed test cases, can be found at [ElementsOfProgrammingInterviews.com/code](http://ElementsOfProgrammingInterviews.com/code). System design problems, and some problems related to probability and discrete mathematics, are conceptual and not meant to be coded.

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

Approximately a quarter of the questions in EPI have a white ninja (☺) or black ninja (☹) designation. White ninja problems are more challenging, and are meant for applicants from whom the bar is higher, e.g., graduate students and tech leads. Black ninja problems are exceptionally difficult, and are suitable for testing a candidate's response to stress, as described on Page 15. Non-ninja questions should be solvable within an hour-long interview and, in some cases, take substantially less time.

Very 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 at Chapter 22.

### ***Level and prerequisites***

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

The review at the start of each chapter is not meant to be comprehensive and if you are not familiar with the material, you should first study it in an algorithms textbook. There are dozens of such texts and our preference is to master one or two good books rather than superficially sample many. *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.

## Part I

### The Interview

# 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 non-technical 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 next 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 following page, and submit it via a personal contact (preferred), or through an online submission process or a campus career fair.
3. Perform an initial phone screening, which often consists of a question-answer session over the phone or video chat with an engineer. You may be asked to submit code via a shared document or an online coding site such as [ideone.com](http://ideone.com) or [collabedit.com](http://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

<b>The Interview Lifecycle, on the preceding page</b> <ul style="list-style-type: none"> <li>– Identify companies, contacts</li> <li>– Résumé preparation <ul style="list-style-type: none"> <li>◊ Basic principles</li> <li>◊ Website with links to projects</li> <li>◊ LinkedIn profile &amp; recommendations</li> </ul> </li> <li>– Résumé submission</li> <li>– Mock interview practice</li> <li>– Phone/campus screening</li> <li>– On-site interview</li> <li>– Negotiating an offer</li> </ul>	<b>At the Interview, on Page 11</b> <ul style="list-style-type: none"> <li>– Don't solve the wrong problem</li> <li>– Get specs &amp; requirements</li> <li>– Construct sample input/output</li> <li>– Work on small examples first</li> <li>– Spell out the brute-force solution</li> <li>– Think out loud</li> <li>– Apply patterns</li> <li>– Test for corner-cases</li> <li>– Use proper syntax</li> <li>– Manage the whiteboard</li> <li>– Be aware of memory management</li> <li>– Get function signatures right</li> </ul>
<b>General Advice, on Page 15</b> <ul style="list-style-type: none"> <li>– Know the company &amp; interviewers</li> <li>– Communicate clearly</li> <li>– Be passionate</li> <li>– Be honest</li> <li>– Stay positive</li> <li>– Don't apologize</li> <li>– Be well-groomed</li> <li>– Mind your body language</li> <li>– Leave perks and money out</li> <li>– Be ready for a stress interview</li> <li>– Learn from bad outcomes</li> <li>– Negotiate the best offer</li> </ul>	<b>Conducting an Interview, on Page 18</b> <ul style="list-style-type: none"> <li>– Don't be indecisive</li> <li>– Create a brand ambassador</li> <li>– Coordinate with other interviewers <ul style="list-style-type: none"> <li>◊ know what to test on</li> <li>◊ look for patterns of mistakes</li> </ul> </li> <li>– Characteristics of a good problem: <ul style="list-style-type: none"> <li>◊ no single point of failure</li> <li>◊ has multiple solutions</li> <li>◊ covers multiple areas</li> <li>◊ is calibrated on colleagues</li> <li>◊ does not require unnecessary domain knowledge</li> </ul> </li> <li>– Control the conversation <ul style="list-style-type: none"> <li>◊ draw out quiet candidates</li> <li>◊ manage verbose/overconfident candidates</li> </ul> </li> <li>– Use a process for recording &amp; scoring</li> <li>– Determine what training is needed</li> <li>– Apply the litmus test</li> </ul>

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

### *The résumé*

It always astonishes us to see candidates who've worked hard for at least four years in school, and often many more in the workplace, spend 30 minutes jotting down random factoids about themselves and calling the result a résumé.

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

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

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

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

### ***Mock interviews***

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

## Strategies For A Great Interview

*The essence of strategy is choosing what not to do.*

— M. E. PORTER

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

### *Approaching the problem*

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

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

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

**Work on small examples:** Consider Problem 21.1 on Page 103, which entails determining which of the 500 doors are open. This problem may seem difficult at first. However, if you start working out which doors are going to be open up to the fifth door, you will see that only Door 1 and Door 4 are open. This may suggest to you that the door is open only if its index is a perfect square. Once you have this epiphany, the proof of its correctness is straightforward. (Keep in mind this approach will not work for all problems you encounter.)

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

a DP solution for a problem (e.g., for Problem [15.4 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 describe the brute-force approach.

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

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

### *Presenting the solution*

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

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

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

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

**Test for corner cases:** For many problems, your general idea may work for most inputs but there may be pathological instances where your algorithm (or your



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

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

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

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

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

### *Know your interviewers & the company*

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

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

In addition to knowing your interviewers, you should know about the company vision, history, organization, products, and technology. You should be ready to talk

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

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

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

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

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

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

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

### *General conversation*

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

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

**Is the candidate passionate about his work?** We always want our colleagues to be excited, energetic, and inspiring to work with. If you feel passionately about your work, and your eyes light up when describing what you've done, it goes a long way in establishing you as a great colleague. Hence when you are asked to describe a project from the past, it is best to pick something that you are passionate about rather than a project that was complex but did not interest you.

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

### *Other advice*

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

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

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

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

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

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

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

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

### *Stress interviews*

Some companies, primarily in the finance industry, make a practice of having one of the interviewers create a stressful situation for the candidate. The stress may be injected technically, e.g., via a ninja problem, or through behavioral means, e.g., the interviewer rejecting a correct answer or ridiculing the candidate. The goal is to see how a candidate reacts to such situations—does he fall apart, become belligerent, or

get swayed easily. The guidelines in the previous section should help you through a stress interview. (Bear in mind you will not know *a priori* if a particular interviewer will be conducting a stress interview.)

### ***Learning from bad outcomes***

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

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

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

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

### ***Negotiating an offer***

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

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

Your HR contact is a professional negotiator, whose fiduciary duty is to the company. He will know and use negotiating techniques such as reciprocity, getting consensus, putting words in your mouth (“don’t you think that’s reasonable?”), as well as threats, to get the best possible deal for the company. (This is what recruiters themselves are evaluated on internally.) The Wikipedia article on negotiation lays bare many tricks we have seen recruiters employ.

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

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

## 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 the morale of the team. On the other hand, discarding good candidates is problematic for a rapidly growing organization. Interviewers also have a moral responsibility not to unfairly crush the interviewee’s dreams and aspirations.

### *Objective*

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

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

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

or insult the candidate over a mistake he made, but such behavior is depressingly common. Outside of a stress interview, the interviewer should work on making the candidate feel positively about the experience, and, by extension, the position and the company.

### *What to ask*

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

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

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

**No single point of failure**—if you are going to ask just one question, you should not pick a problem where the candidate passes the interview if and only if he gets one particular insight. The best candidate may miss a simple insight, and a mediocre candidate may stumble across the right idea. There should be at least two or three opportunities for the candidates to redeem themselves. For example, problems that can be solved by dynamic programming can almost always be solved through a greedy algorithm that is fast but suboptimum or a brute-force algorithm that is slow but optimum. In such cases, even if the candidate cannot get the key insight, he can still demonstrate some problem solving abilities. Problem 6.2 on Page 50 exemplifies this type of question.

**Multiple possible solutions**—if a given problem has multiple solutions, the chances of a good candidate coming up with a solution increases. It also gives the interviewer more freedom to steer the candidate. A great candidate may finish

with one solution quickly enough to discuss other approaches and the trade-offs between them. For example, Problem 11.6 on Page 70 can be solved using a hash table or a bit array; the best solution makes use of binary search.

**Cover multiple areas**—even if you are responsible for testing the candidate on algorithms, you could easily pick a problem that also exposes some aspects of design and software development. For example, Problem 18.2 on Page 96 tests candidates on concurrency as well as data structures. Problem 17.1 on Page 92 requires knowledge of both dynamic programming and probability.

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

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

### *Conducting the interview*

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

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

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

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

### *Scoring and reporting*

At the end of an interview, the interviewers usually have a good idea of how the candidate scored. However, 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.

## Problem Solving Patterns

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

— A. EINSTEIN

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

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

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

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

These patterns are summarized in Table 4.1 on the facing page, Table 4.2 on Page 28, and Table 4.3 on Page 36, respectively.

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

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

### ***Data structure patterns***

A data structure is a particular way of storing and organizing related data items so that they can be manipulated efficiently. Usually the correct selection of data structures is key to designing a good algorithm. Different data structures are suited to different applications; some are highly specialized. For example, heaps are particularly well-suited for algorithms that merge sorted data streams, while compiler implementations usually use hash tables to look up identifiers.

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

#### PRIMITIVE TYPES

You should be comfortable with the basic types (chars, integers, doubles, etc.), their variants (unsigned, long, etc.), and operations on them (bitwise operators, comparison, etc.). Don't forget that the basic types differ among programming languages. For example, Java has no unsigned integers, and the number of bits in an integer is compiler- and machine-dependent in C.

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

Another approach, which may run faster on some inputs, is based on computing  $y = x \& \sim(x-1)$ , 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  $x \oplus y$ . The time complexity is  $O(s)$ , where  $s$  is the number of bits set to 1 in  $x$ .

In practice if the computation is done repeatedly, the most efficient approach would be to create a lookup table. In this case, we could use a 256 entry integer-valued array  $P$  such that  $P[i]$  is the number of bits set to 1 in  $i$ . If  $x$  is 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 AND STRINGS

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

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

The key to the solution is to maintain two regions on opposite sides of the array that meet the requirements, and expand these regions one element at a time.

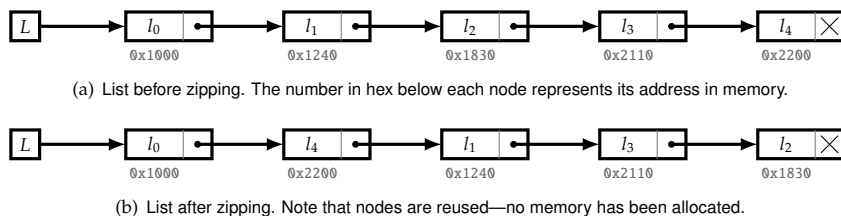
## LISTS

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

A list is similar to an array in that it contains objects in a linear order. The key differences are that inserting and deleting elements in a list has time complexity  $O(1)$ . On the other hand, obtaining the  $k$ -th element in a list is expensive, having  $O(n)$

time complexity. Lists are usually building blocks of more complex data structures. However, they can be the subject of tricky problems in their own right, as illustrated by the following:

Given a singly linked list  $\langle l_0, l_1, l_2, \dots, l_{n-1} \rangle$ , define the “zip” of the list to be  $\langle l_0, l_{n-1}, l_1, l_{n-2}, \dots \rangle$ . 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.



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

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

## HEAPS

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

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

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

## HASH TABLES

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

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

## BINARY SEARCH TREES

Binary search trees (BSTs) are used to store objects that are comparable. The underlying idea is to organize the objects in a binary tree in which the nodes satisfy the BST property: the key stored at any node is greater than or equal to the keys stored in its left subtree and less than or equal to the keys stored in its right subtree. Insertion and deletion can be implemented so that the height of the BST is  $O(\log n)$ , leading to fast ( $O(\log n)$ ) lookup and update times. AVL trees and red-black trees are BST implementations that support this form of insertion and deletion.

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

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

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

### Other data structures

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

- *Skip lists*, which store a set of comparable items using a hierarchy of sorted linked lists. Lists higher in the hierarchy consist of increasingly smaller subsequences of the items. Skip lists implement the same functionality as balanced BSTs, but are simpler to code and faster, especially when used in a concurrent context.
- *Treaps*, which are a combination of a BST and a heap. When an element is inserted into a treap it is assigned a random key that is used in the heap organization. The advantage of a treap is that it is height-balanced with high probability and the insert and delete operations are considerably simpler than for deterministic height-balanced trees such as AVL and red-black trees.
- *Fibonacci heaps*, which consist of a series of trees. Insert, find minimum, decrease key, and merge (union) run in amortized constant time; delete and delete-minimum take  $O(\log n)$  time. In particular Fibonacci heaps can be used to reduce the time complexity of Dijkstra's shortest path algorithm from  $O((|E| + |V|) \log |V|)$  to  $O(|E| + |V| \log |V|)$ .
- *Disjoint-set data structures*, which are used to manipulate subsets. The basic operations are union (form the union of two subsets), and find (determine which set an element belongs to). These are used in a number of algorithms, notably in tracking connected components in an undirected graph and Kruskal's algorithm for the minimum spanning tree. We use the disjoint-set data structure to solve the offline minimum problem.
- *Tries*, which are a tree-based data structure used to store strings. Unlike BSTs, nodes do not store keys; instead, the node's position in the tree determines the key it is associated with. Tries can have performance advantages with respect to BSTs and hash tables; they can also be used to solve the longest matching prefix problem.

### 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 programming	Compute solutions for smaller instances of a given problem and use these solutions to construct a solution to the problem. Cache for performance.
The greedy method	Compute a solution in stages, making choices that are locally optimum at step; these choices are never undone.
Incremental improvement	Quickly build a feasible solution and improve its quality with small, local updates.
Elimination	Identify and rule out potential solutions that are sub-optimal or dominated by other solutions.
Parallelism	Decompose the problem into subproblems that can be solved independently on different machines.
Caching	Store computation and later look it up to save work.
Randomization	Use randomization within the algorithm to reduce complexity.
Approximation	Efficiently compute a suboptimum solution that is of acceptable quality.
State	Identify an appropriate notion of state.

SORTING

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

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



Sorting is not appropriate when an  $O(n)$  (or better) algorithm is possible. Furthermore, sorting can obfuscate the problem. For example, given an array  $A$  of numbers, if we are to determine the maximum of  $A[i] - A[j]$ , for  $i < j$ , sorting destroys the order and complicates the problem.

## RECURSION

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

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

- $x$  where  $x$  is a character, for simplicity assumed to be a lower-case letter (matches the string “ $x$ ”).
- $.$  (matches any string of length 1).
- $x^*$  (matches the string consisting of zero or more occurrences of the character  $x$ ).
- $.*$  (matches the string consisting of zero or more of any characters).
- $r_1r_2$  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  $s_2$ ).

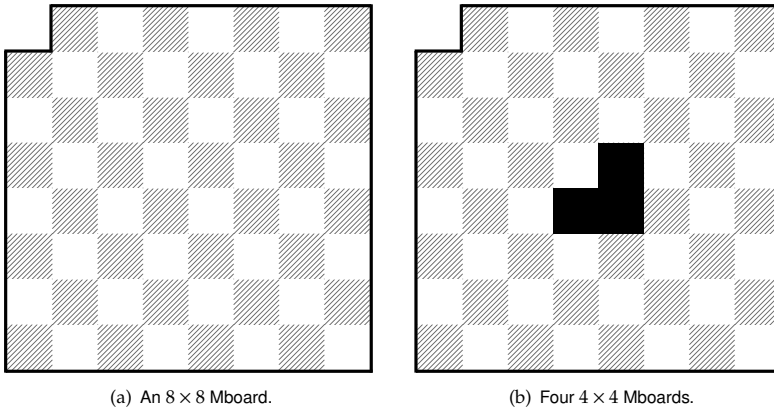
This problem can be solved by checking a number of cases based on the first one or two characters of the matching expression, and recursively matching the rest of the string.

## 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 15; we illustrate the basic idea below.

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

Divide and conquer is a good strategy for this problem. Instead of the  $8 \times 8$  Mboard, let's consider an  $n \times n$  Mboard. A  $2 \times 2$  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.



**Figure 4.2:** Mutilated chessboards.

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

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

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

Other examples of divide and conquer include counting the number of pairs of elements in an array that are out of sorted order 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 one ending with a 3 point play, and a combination ending with a 3 point play is different from one ending with a 7 point play, etc.

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

Implementing the recursion naïvely results in multiple calls to the same sub-stance. 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.

#### THE GREEDY METHOD

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

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

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

### INCREMENTAL IMPROVEMENT

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

As an example consider a department with  $n$  graduate students and  $n$  professors. Each student begins with a rank ordered preference list of the professors based on how keen he is to work with each of them. Each professor has a similar preference list of students. Suppose you were asked to devise an algorithm which takes as input the preference lists and outputs a one-to-one pairing of students and advisers in which there are no student-adviser pairs  $(s_0, a_0)$  and  $(s_1, a_1)$  such that  $s_0$  prefers  $a_1$  to  $a_0$  and  $a_1$  prefers  $s_0$  to  $s_1$ .

Here is an algorithm for this problem in the spirit of incremental improvement. Each student who does not have an adviser “proposes” to the most-preferred professor to whom he has not yet proposed. Each professor then considers all the students who have proposed to him and says to the student in this set he most prefers “I accept you”; he says “no” to the rest. The professor is then provisionally matched to a student; this is the candidate solution. In each subsequent round, each student who does not have an adviser proposes to the professor to whom he has not yet proposed who is highest on his preference list. He does this regardless of whether the professor has already been matched with a student. The professor once again replies with a single accept, rejecting the rest. In particular, he may leave a student with whom he is currently paired.

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

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

### ELIMINATION

One common approach to designing an efficient algorithm is to use elimination—that is to identify and rule out potential solutions that are suboptimal or dominated by other solutions. Binary search, which is the subject of a number of problems in Chapter 11, uses elimination. Solution 11.5 on Page 146, where we use elimination to compute the square root of a real number, is especially instructive. Below we consider a fairly sophisticated application of elimination.

Suppose you have to build a distributed storage system. A large number,  $n$ , of users will share data on your system, which consists of  $m$  servers, numbered from 0 to  $m-1$ . One way to distribute users across servers is to assign the user with login ID  $l$  to

the server  $h(l) \bmod m$ , where  $h()$  is a hash function. If the hash function does a good job, this approach distributes users uniformly across servers. However, if certain users require much more storage than others, some servers may be overloaded while others idle.

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

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

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

For the case of 10000 users and 100 servers, the DP algorithm took over an hour; the approach using binary search for  $b$  with greedy assignment took 0.1 seconds.

#### PARALLELISM

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

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

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

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

#### CACHING

Caching is a great tool whenever computations are repeated. For example, the central idea behind dynamic programming is caching results from intermediate computations. Caching is also extremely useful when implementing a service that is expected to respond to many requests over time, and many requests are repeated. Workloads on web services exhibit this property.

## RANDOMIZATION

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

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

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

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

## APPROXIMATION

In the real-world it is routine to be given a problem that is difficult to solve exactly, either because of its intrinsic complexity, or the complexity of the code required. Developers need to recognize such problems, and be ready to discuss alternatives with the author of the problem. In practice a solution that is “close” to the optimum solution is usually perfectly acceptable.

Let  $\{A_0, A_1, \dots, A_{n-1}\}$  be a set of  $n$  cities, as in Figure 4.3 on the facing page. Suppose we need to choose a subset of  $A$  to locate warehouses. Specifically, we want to choose



There may be multiple ways in which state can be defined, all of which lead to correct algorithms. When computing the max-difference (Problem 6.2 on Page 50), we could use the values of the elements at all prior indices as the state when we iterate through the array. Of course, this is inefficient, since all we really need is the minimum value.

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

**Abstract analysis patterns**

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

**Table 4.3:** Abstract analysis techniques.

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

CASE ANALYSIS

In case analysis a problem is divided into a number of separate cases, and analyzing each such case individually suffices to solve the initial problem. Cases do not have



to be mutually exclusive; however, they must be exhaustive, that is cover all possibilities. For example, to prove that for all  $n$ ,  $n^3 \bmod 9$  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  $S_1, \dots, S_5$  of  $S$ , sort each subset, and then sort  $\{\max S_1, \dots, \max S_5\}$ . 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.

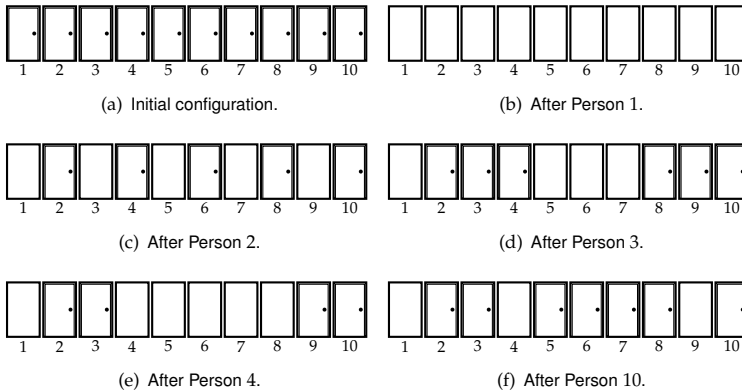
#### SMALL EXAMPLES

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

It is difficult to solve this problem using an abstract approach, e.g., introducing Boolean variables for the state of each door and a state update function. However if you try the same problem with 1, 2, 3, 4, 10, and 20 doors, it takes a short time to see that the doors that remain open are 1, 4, 9, 16, ..., regardless of the total number of doors. The 10 doors case is illustrated in Figure 4.4 on the next page. Now the pattern is obvious—the doors that remain open are those corresponding to the perfect squares. Once you make this connection, it is easy to prove it for the general case. Hence the total number of open doors is  $\lfloor \sqrt{500} \rfloor = 22$ .

#### ITERATIVE REFINEMENT OF A BRUTE-FORCE SOLUTION

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



**Figure 4.4:** Progressive updates to 10 doors.

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  $B[0] \leq B[1] \geq B[2] \leq B[3] \geq B[4] \leq B[5] \geq \dots$ .

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  $(A[1], A[2]), (A[3], A[4]), \dots$ . 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     for (size_t i = 1; i < A->size(); ++i) {
3         if (((i & 1) == 0 && (*A)[i - 1] < (*A)[i]) ||
4             ((i & 1) == 1 && (*A)[i - 1] > (*A)[i])) {
5             swap((*A)[i - 1], (*A)[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  $s = abdbabcabc$  and if, starting backwards, we have a partial match up to  $abcbabc$  that does not result in a full match, we know that the next possible matching offset has to be at least three positions ahead (where we can match the second  $abc$  from the partial match).

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

Many other sophisticated algorithms can be developed in this fashion. As another example, the brute-force solution to computing the maximum subarray sum for an integer array of length  $n$  is to compute the sum of all subarrays, which has  $O(n^3)$  time complexity. This can be improved to  $O(n^2)$  by precomputing the sums of all the prefixes of the given arrays; this allows the sum of a subarray to be computed in  $O(1)$  time. The natural divide and conquer algorithm has an  $O(n \log n)$  time complexity. Finally, one can observe that a maximum subarray must end at one of  $n$  indices, and the maximum subarray sum for a subarray ending at index  $i$  can be computed from previous maximum subarray sums, which leads to an  $O(n)$  algorithm. Details are presented on Page 80.

## REDUCTION

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

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

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 prob-

lem. This shows that the given problem is difficult, which justifies heuristics and approximate solutions. Such scenarios are described in more detail in Chapter 17.

GRAPH MODELING

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

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

Table 4.4: Exchange rates for seven major currencies.

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

We can model the problem with a graph where currencies correspond to vertices, exchanges correspond to edges, and the edge weight is set to the logarithm of the exchange rate. If we can find a cycle in the graph with a positive weight, we would have found such a series of exchanges. Such a cycle can be solved using the Bellman-Ford algorithm.

WRITE AN EQUATION

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

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

The binomial coefficients satisfy the *addition formula*:

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

This identity leads to a straightforward recursion for computing  $\binom{n}{k}$  which avoids the problems described above. DP has to be used to achieve good time complexity.

#### VARIATION

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

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

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

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

It is not necessary to find a coloring that maximizes the number of diverse edges. All that is needed is a coloring in which the set of diverse edges is maximal with respect to single vertex flips. Such a coloring can be computed efficiently.

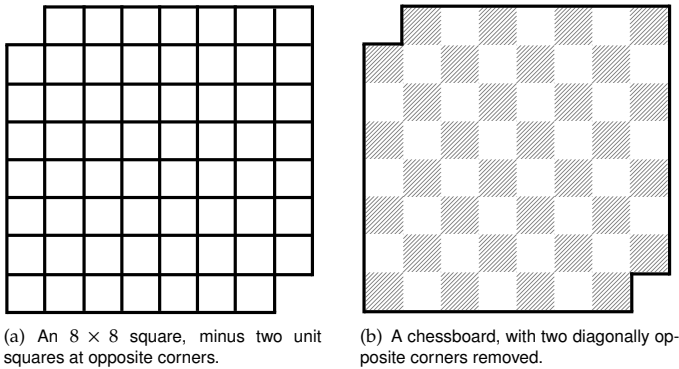
#### INVARIANTS

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

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

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

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



**Figure 4.5:** Invariant analysis exploiting auxiliary elements.

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

It is possible to prove impossibility without appealing to square colors. Specifically, orient the board with the missing pieces on the lower right and upper left. An impossibility proof exists based on a case-analysis for each column on the height of the highest domino that is parallel to the base. However, the proof given above is much simpler.

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



**Figure 4.6:** Coins in a row.

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

Invariant analysis can be used with symmetry to solve very difficult problems,

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

### Complexity Analysis

The run time of an algorithm depends on the size of its input. One common approach to capture the run time dependency is by expressing asymptotic bounds on the worst-case run time as a function of the input size. Specifically, the run time of an algorithm on an input of size  $n$  is  $O(f(n))$  if, for sufficiently large  $n$ , the run time is not more than  $f(n)$  times a constant. The big- $O$  notation simply indicates an upper bound; if the run time is asymptotically proportional to  $f(n)$ , the complexity is written as  $\Theta(f(n))$ . (Note that the big- $O$  notation is widely used where sometimes  $\Theta$  is more appropriate.) The notation  $\Omega(f(n))$  is used to denote an asymptotic lower bound of  $f(n)$  on the time complexity of an algorithm.

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

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

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

Several of our problems call for an algorithm that uses  $O(1)$  space. Conceptually, the memory used by such an algorithm should not depend on the size of the input instance. Specifically, it should be possible to implement the algorithm without dynamic memory allocation (explicitly, or indirectly, e.g., through library routines).

Furthermore, the maximum depth of the function call stack should also be a constant, independent of the input. The standard algorithm for depth-first search of a graph is an example of an algorithm that does not perform any dynamic allocation, but uses the function call stack for implicit storage—its space complexity is not  $O(1)$ .

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

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

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



Part II

Problems

# Primitive Types

*Representation is the essence of programming.*

— “The Mythical Man Month,”

F. P. BROOKS, 1975

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

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

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

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

## 5.1 COMPUTING PARITY

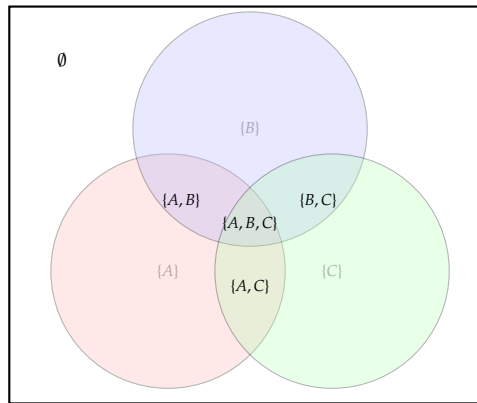
The parity of a sequence of bits is 1 if the number of 1s in the sequence is odd; otherwise, it is 0. Parity checks are used to detect single bit errors in data storage and communication. It is fairly straightforward to write code that computes the parity of a single 64-bit nonnegative integer.

**Problem 5.1:** How would you go about computing the parity of a very large number of 64-bit nonnegative integers? pg. 112

## 5.2 THE POWER SET

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

next page.



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

**Problem 5.2:** Implement a method that takes as input a set  $S$  of distinct elements, and prints the power set of  $S$ . Print the subsets one per line, with elements separated by commas. pg. 113

### 5.3 STRING AND INTEGER CONVERSIONS

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

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

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

**Problem 5.3:** Implement string/integer inter-conversion functions. Use the following function signatures: `String intToString(int x)` and `int stringToInt(String s)`. pg. 114

### 5.4 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 5.4:** Design an efficient algorithm for computing the GCD of two numbers without using multiplication, division or the modulus operators. pg. 115

5.5 COMPUTING  $x/y$  (🧠)

**Problem 5.5:** 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. 116

## Arrays and Strings

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

—“Intelligent Machinery,”

A. M. TURING, 1948

### Arrays

The simplest data structure is the *array*, which is a contiguous block of memory. Given an array  $A$  which holds  $n$  objects,  $A[i]$  denotes the  $i + 1$ -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.

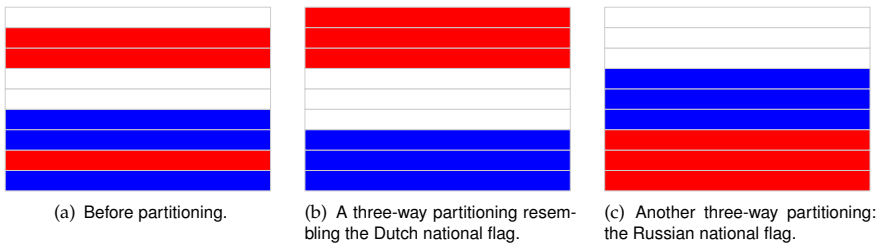
Insertion of an object into a full array can be handled by allocating a new array with additional memory and copying over the entries from the original array. This makes the worst-case time of insertion high but if the new array has, for example, twice the space of the original array, the average time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

#### 6.1 DUTCH NATIONAL FLAG

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

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

When an array consists of entries from a small set of keys, e.g.,  $\{0, 1, 2\}$ , one way to sort it is to count the number of occurrences of each key. Consequently, enumerate the keys in sorted order and write the corresponding number of keys to the array. If a BST is used for counting, the time complexity of this approach is  $O(n \log k)$ , where  $n$  is the array length and  $k$  is the number of keys. This is known as counting sort. Counting sort, as just described, does not differentiate among different objects with the same key value. This problem is concerned with a special case of counting sort when entries are objects rather than keys. Problem 13.2 on Page 74 addresses the general problem.



**Figure 6.1:** Illustrating the Dutch national flag problem.

**Problem 6.1:** Write a function that takes an array  $A$  and an index  $i$  into  $A$ , and rearranges the elements such that all elements less than  $A[i]$  appear first, followed by elements equal to  $A[i]$ , followed by elements greater than  $A[i]$ . Your algorithm should have  $O(1)$  space complexity and  $O(|A|)$  time complexity. pg. 117

## 6.2 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. 118

## 6.3 GENERALIZATIONS OF MAX DIFFERENCE (🔒)

Problem 6.2 on the facing page, which is concerned with computing  $\max_{0 \leq i < j \leq n-1} (A[j] - A[i])$ , 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  $i_0 < j_0 < i_1 < j_1 < \dots < i_{k-1} < j_{k-1}$ . Here  $k$  is a fixed input parameter.
- (3.) Repeat Problem (2.) when  $k$  can be chosen to be any value from 0 to  $\lfloor n/2 \rfloor$ .

pg. 119

## 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 12) and dynamic programming (Page 80).

### 6.4 REVERSE ALL THE WORDS IN A SENTENCE

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

**Problem 6.4:** Implement a function for reversing the words in a string. Your function should use  $O(1)$  space. pg. 120

### 6.5 REGULAR EXPRESSION MATCHING

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

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

An extended simple regular expression (ESRE) is an SRE, an SRE prepended with the metacharacter `^`, an SRE ended with the metacharacter `$`, or an SRE prepended with `^` and ended with `$`. The previous SRE examples are ESREs, as are `^a`, `aW$`, and `^aW.9*$`.

First we define what it means for an SRE  $r$  to strictly match a string  $s$ . Recall  $s^k$  denotes the  $k$ -th suffix of  $s$ , i.e., the string resulting from deleting the first  $i$  characters from  $s$ . For example, if  $s = \text{aWaW9W9}$ , then  $s^0 = \text{aWaW9W9}$ , and  $s^2 = \text{aW9W9}$ .

- If  $r$  begins with an alphanumeric character and the next character in  $r$  is not star, then  $r$  strictly matches  $s$  if that same character is at the start of  $s$ , and  $r^1$  strictly matches  $s^1$ .

- If  $r$  begins with an alphanumeric character and the next character in  $r$  is star, then  $r$  strictly matches  $s$  if  $s$  can be written as  $s_1$  concatenated by  $s_2$ , where  $s_1$  consists of zero or more of the same character, and  $s_2$  strictly matches  $r^2$ .
- If  $r$  begins with dot and the next character in  $r$  is not star, then  $r$  strictly matches  $s$  if  $r^1$  strictly matches  $s^1$ .
- If  $r$  begins with dot and the next character in  $r$  is star, then  $r$  strictly matches  $s$  if  $s$  can be written as  $s_1$  concatenated with  $s_2$ , where  $s_1$  is of length 0 or more, and  $r^2$  strictly matches  $s_2$ .

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

An ESRE  $r$  beginning with  $^$  matches  $s$  if there is a prefix  $s_1$  of  $s$  such that  $r$  strictly matches  $s_1$ . An ESRE  $r$  ending with  $$$  matches  $s$  if there is a suffix  $s_2$  of  $s$  such that  $r$  strictly matches  $s_2$ .

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

**Problem 6.5:** Design an algorithm that takes a string  $s$  and a string  $r$ , assumed to be a well-formed ESRE, and checks if  $r$  matches  $s$ . pg. 121



## 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 7.1. There are many variants of linked lists, e.g., in a *doubly linked list*, each node has a link to its predecessor; similarly, a sentinel node or a self-loop can be used instead of null. The structure of a doubly linked list is given in Figure 7.2. Since lists can be defined recursively, recursion is a natural candidate for list manipulation.

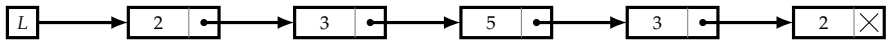


Figure 7.1: Example of a singly linked list.

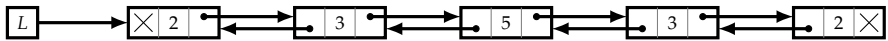


Figure 7.2: Example of a doubly linked list.

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

```

1 template <typename T>
2 struct node_t {
3     T data;

```

```
4 shared_ptr<node_t<T>> next;  
5 };
```

7.1 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 7.3.

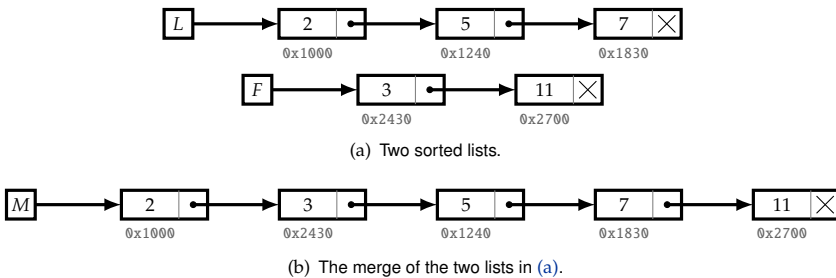


Figure 7.3: Merging sorted lists.

**Problem 7.1:** Write a function that takes  $L$  and  $F$ , and returns the merge of  $L$  and  $F$ . Your code should use  $O(1)$  additional storage—it should reuse the nodes from the lists provided as input. Your function should use  $O(1)$  additional storage, as illustrated in Figure 7.3. The only field you can change in a node is next. *pg. 122*

7.2 CHECKING FOR CYCLICITY

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

**Problem 7.2:** Given a reference to the head of a singly linked list  $L$ , how would you determine whether  $L$  ends in a null or reaches a cycle of nodes? Write a function that returns null if there does not exist a cycle, and the reference to the start of the cycle if a cycle is present. (You do not know the length of the list in advance.) *pg. 123*

7.3 OVERLAPPING LISTS—NO LISTS HAVE CYCLE

Given two singly linked lists,  $L1$  and  $L2$ , there may be list nodes that are common to both  $L1$  and  $L2$ . (This may not be a bug—it may be desirable from the perspective of reducing memory footprint, as in the flyweight pattern, or maintaining a canonical form.) For example,  $L1$  and  $L2$  in Figure 7.4 on the next page overlap at Node  $I$ .

**Problem 7.3:** Let  $h1$  and  $h2$  be the heads of lists  $L1$  and  $L2$ , respectively. Assume that  $L1$  and  $L2$  are well-formed, that is each consists of a finite sequence of nodes. (In

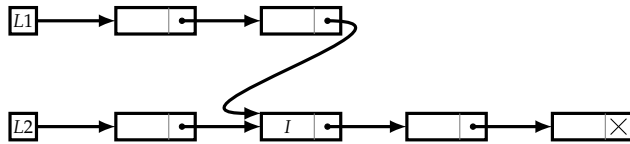


Figure 7.4: Example of overlapping lists.

particular, neither list has a cycle.) How would you determine if there exists a node  $r$  reachable from both  $h1$  and  $h2$  by following the next fields? If such a node exists, find the node that appears earliest when traversing the lists. You are constrained to use no more than constant additional storage. pg. 125

#### 7.4 REVERSING A SINGLY LINKED LIST

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

**Problem 7.4:** Give a linear time non-recursive function that reverses a singly linked list. The function should use no more than constant storage beyond that needed for the list itself. pg. 126

#### 7.5 COPYING A POSTINGS LIST (☹️)

In a “postings list” each node has a data field, a field for the next pointer, and a jump field—the jump field points to any other node. The last node in the postings list has next set to null; all other nodes have non-null next and jump fields. For example, Figure 7.5 is a postings list with four nodes.

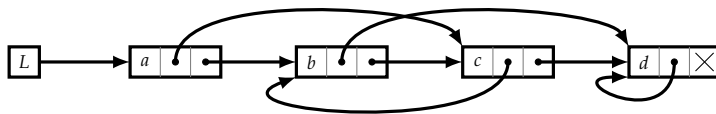


Figure 7.5: A postings list.

**Problem 7.5:** 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. 127

# Stacks and Queues

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

— “The Art of Computer Programming, Volume 1,”  
D. E. KNUTH, 1997

## Stacks

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

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

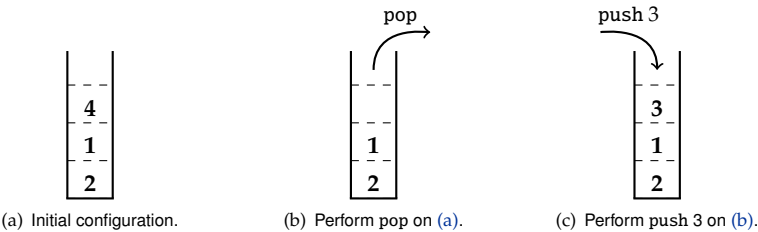


Figure 8.1: Operations on a stack.

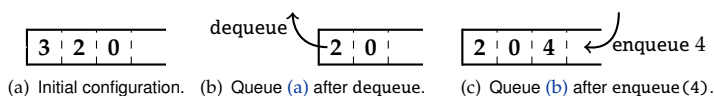
### 8.1 STACK WITH MAX OPERATION

**Problem 8.1:** Design a stack that supports a **max** operation, which returns the maximum value stored in the stack, and throws an exception if the stack is empty. Assume elements are comparable. All operations must be  $O(1)$  time. You can use  $O(n)$  additional space, beyond what is required for the elements themselves. pg. 128

## Queues

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

A queue can be implemented using a linked list, in which case these operations have  $O(1)$  time complexity. Other operations can be added, such as head (which returns the item at the start of the queue without removing it), and tail (which returns the item at the end of the queue without removing it). A queue can also be implemented using an array; see Problem 8.3 on the following page for details.



**Figure 8.2:** 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*.

## 8.2 PRINTING A BINARY TREE IN LEVEL ORDER

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

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

**Problem 8.2:** Given the root node  $r$  of a binary tree, print all the keys in level order at  $r$  and its descendants. Specifically, the nodes should be printed in order of their level, with all keys at a given level in a single line, and the next line corresponding to keys at the next level. 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 9.1 on Page 59.

pg. 131

### 8.3 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 8.3:** 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. *pg. 132*

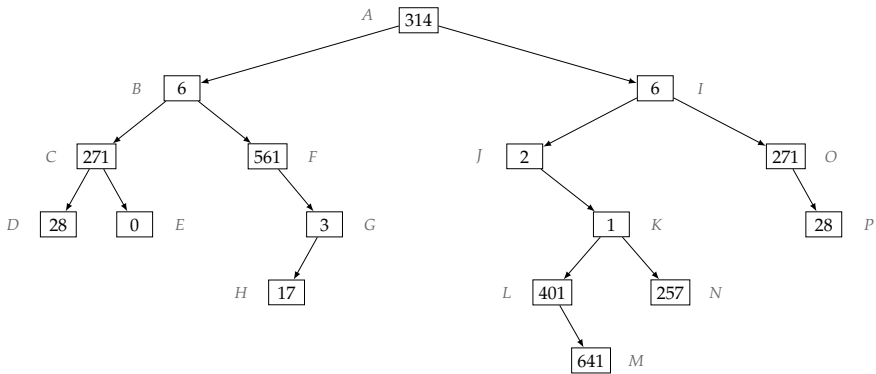
## Binary Trees

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

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

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

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



**Figure 9.1:** Example of a binary tree.

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

```

1 template <typename T>
2 struct BinaryTree {
3     T data;
4     unique_ptr<BinaryTree<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. Often, but not always, the node has a parent field (which is null for the root). Observe that for any node  $n$  there exists a unique sequence of nodes from the root to  $n$  with each subsequent node being a child of the previous node. This sequence is sometimes referred to as the *search path* from the root to  $n$ .

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

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

As concrete examples of these concepts, consider the binary tree in Figure 9.1 on the 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 or same level, and in which every parent has two children. A *complete binary tree* is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible. (This terminology is not universal, e.g., some authors use complete binary tree where we write perfect binary tree.) It is straightforward to prove using induction that the number of non-leaf nodes in a full binary tree is one less than the number of leaves. A perfect binary tree of height  $h$  contains exactly  $2^{h+1} - 1$  nodes, of which  $2^h$  are leaves. A complete binary tree on  $n$  nodes has height  $\lceil \lg n \rceil$ .

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

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

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

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



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

```

1  template <typename T>
2  void inorder_traversal(const unique_ptr<BinaryTree<T>>& root) {
3      auto* n = root.get();
4      while (n) {
5          if (n->left.get()) {
6              // Find the predecessor of n.
7              auto* pre = n->left.get();
8              while (pre->right.get() && pre->right.get() != n) {
9                  pre = pre->right.get();
10             }
11
12             // Process the successor link.
13             if (pre->right.get()) { // pre->right.get() == n.
14                 // Revert the successor link if predecessor's successor is n.
15                 pre->right.release();
16                 cout << n->data << endl;
17                 n = n->right.get();
18             } else { // if predecessor's successor is not n.
19                 pre->right.reset(n);
20                 n = n->left.get();
21             }
22         } else {
23             cout << n->data << endl;
24             n = n->right.get();
25         }
26     }
27 }

```

The term *tree* is overloaded, which can lead to confusion; see Page 87 for an overview of the common variants.

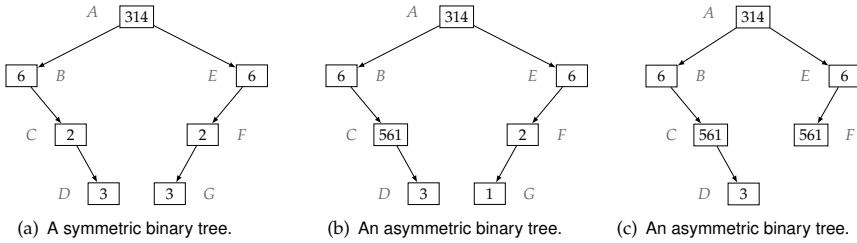
## 9.1 SYMMETRIC BINARY TREE

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

**Problem 9.1:** Write a function that takes as input the root of a binary tree and returns true or false depending on whether the tree is symmetric. pg. 133

## 9.2 INORDER TRAVERSAL WITH $O(1)$ SPACE (🧐)

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



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

**Problem 9.2:** 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. 133

### 9.3 SUCCESSOR IN A BINARY TREE

The successor of a node  $n$  in a binary tree is the node  $s$  that appears immediately after  $n$  in an inorder walk. For example, in Figure 9.1 on Page 59, the successor of Node G is Node A, and the successor of Node A is Node J.

**Problem 9.3:** 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. 134

### 9.4 LOWEST COMMON ANCESTOR IN A BINARY TREE

Any two nodes in a binary tree have a common ancestor, namely the root. The lowest common ancestor (LCA) of any two nodes in a binary tree is the node furthest from the root that is an ancestor of both nodes. For example, the LCA of M and N in Figure 9.1 on Page 59 is K.

**Problem 9.4:** 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. 135

## Heaps

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

— “Fibonacci heaps and their uses,”

M. L. FREDMAN AND R. E. TARJAN, 1987

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

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

### 10.1 MERGING SORTED FILES

You are given 500 files, each containing stock trade information for an S&P 500 company. 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 10.1:** 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. 136

10.2 SORT *k*-INCREASING-DECREASING ARRAY

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

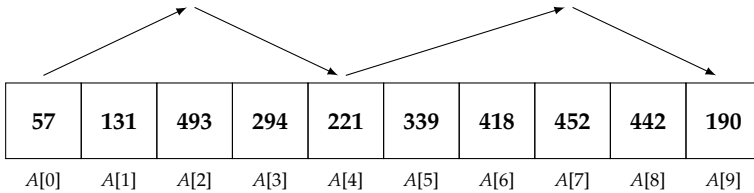


Figure 10.1: A 4-increasing-decreasing array.

**Problem 10.2:** Design an efficient algorithm for sorting a *k*-increasing-decreasing array. You are given another array of the same size that the result should be written to, and you can use  $O(k)$  additional storage. pg. 137

10.3 CLOSEST STARS

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

**Problem 10.3:** How would you compute the *k* stars which are closest to the Earth? You have only a few megabytes of RAM. pg. 137

10.4 APPROXIMATE SORT

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

**Problem 10.4:** The input consists of a very long sequence of numbers. Each number is at most *k* positions away from its correctly sorted position. Design an algorithm that outputs the numbers in the correct order and uses  $O(k)$  storage, independent of the number of elements processed. pg. 139

10.5 GENERATING NUMBERS OF THE FORM  $a + b\sqrt{2}$  (☹)

Let  $S_q$  be the set of real numbers of the form  $a + b\sqrt{q}$ , where *a* and *b* are nonnegative integers, and *q* is an integer which is not the square of another integer. Such sets

have special properties, e.g., they are closed under addition and multiplication. The first few numbers of this form are given in Figure 10.2.

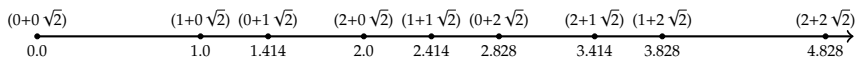
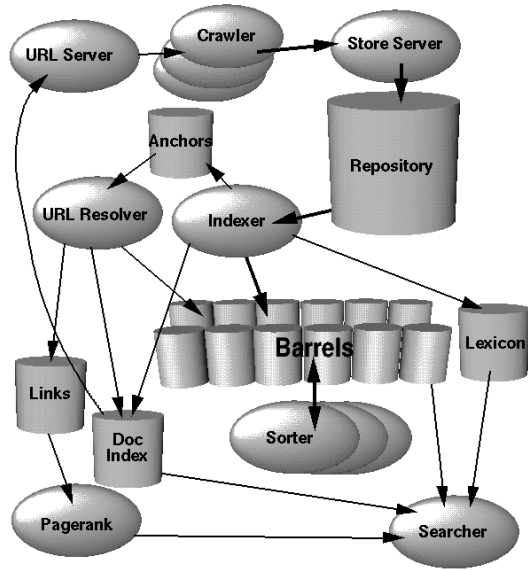


Figure 10.2: Points of the form  $a + b\sqrt{2}$ .

**Problem 10.5:** Design an algorithm for efficiently computing the  $k$  smallest real numbers of the form  $a + b\sqrt{2}$  for nonnegative integers  $a$  and  $b$ . pg. 139

## Searching



—“The Anatomy of A Large-Scale Hypertextual Web Search Engine,”

S. M. BRIN AND L. PAGE, 1998

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

### Binary search

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

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

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

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

```

1 int bsearch(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.

### 11.1 SEARCH A SORTED ARRAY FOR FIRST OCCURRENCE OF $k$

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

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

Figure 11.1: A sorted array with repeated elements.

**Problem 11.1:** Write a method that takes a sorted array  $A$  and a key  $k$  and returns the index of the *first* occurrence of  $k$  in  $A$ . Return  $-1$  if  $k$  does not appear in  $A$ . For example, when applied to the array in Figure 11.1 your algorithm should return 3 if  $k = 108$ ; if  $k = 285$ , your algorithm should return 6. pg. 141

### 11.2 SEARCH FOR A PAIR IN AN ABS-SORTED ARRAY (🧐)

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

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

Figure 11.2: An abs-sorted array.

**Problem 11.2:** Design an algorithm that takes an abs-sorted array  $A$  and a number  $k$ , and returns a pair of indices of elements in  $A$  that sum up to  $k$ . For example, if the



input to your algorithm is the array in Figure 11.2 on the preceding page and  $k = 167$ , your algorithm should output (3, 7). Output  $(-1, -1)$  if there is no such pair. *pg. 142*

### 11.3 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  $\langle A[i], A[i+1], \dots, A[n-1], A[0], A[1], \dots, A[i-1] \rangle$  is sorted in increasing order, as illustrated in Figure 11.3.

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

Figure 11.3: A cyclically sorted array.

**Problem 11.3:** Design an  $O(\log n)$  algorithm for finding the position of the smallest element in a cyclically sorted array. Assume all elements are distinct. For example, for the array in Figure 11.3, your algorithm should return 4. *pg. 144*

### 11.4 SEARCHING IN TWO SORTED ARRAYS (🔒)

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

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

**Problem 11.4:** 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. 145*

### 11.5 COMPUTING SQUARE ROOTS

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

**Problem 11.5:** Implement a function which takes as input a floating point variable  $x$  and returns  $\sqrt{x}$ . *pg. 146*

### Searching unsorted arrays

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

#### 11.6 FINDING A MISSING ELEMENT

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

**Problem 11.6:** 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. 147*

#### 11.7 MAJORITY FIND (👤)

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

**Problem 11.7:** You are reading a sequence of words from a very long stream. You know *a priori* that more than half the words are repetitions of a single word  $w$  (the “majority element”) but the positions where  $w$  occurs are unknown. Design an algorithm that makes a single pass over the stream and uses only a constant amount of memory to identify  $w$ . *pg. 148*

## Hash Tables

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

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

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

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

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

### 12.1 ANAGRAMS

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

**Problem 12.1:** 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. 148*

## 12.2 ANONYMOUS LETTER

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

**Problem 12.2:** 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. 149*

## 12.3 LINE THROUGH THE MOST POINTS (🧐)

**Problem 12.3:** 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. 150*

## Sorting

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

—“Quicksort,”

C. A. R. HOARE, 1962

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

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

The heap data structure is discussed in detail in Chapter 10. Briefly, a max-heap (min-heap) stores keys drawn from an ordered set. It supports  $O(\log n)$  inserts and  $O(1)$  time lookup for the maximum (minimum) element; the maximum (minimum) key can be deleted in  $O(\log n)$  time. Heaps can be helpful in sorting problems, as illustrated by Problems 10.1 on Page 63, 10.2 on Page 64, and 10.4 on Page 64.

### 13.1 VARIABLE LENGTH SORT

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

**Problem 13.1:** Sort lines of a text file that has one million lines such that the average length of a line is 100 characters but the longest line is one million characters long.

pg. 152

## 13.2 COUNTING SORT (🧠)

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

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

**Problem 13.2:** You are given an array of  $n$  Person objects. Each Person object has a field `key`. Rearrange the elements of the array so that Person objects with equal keys appear together. The order in which distinct keys appear is not important. Your algorithm must run in  $O(n)$  time and  $O(k)$  additional space. How would your solution change if keys have to appear in sorted order? pg. 153

## 13.3 INTERSECT TWO SORTED ARRAYS

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

**Problem 13.3:** 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. 154

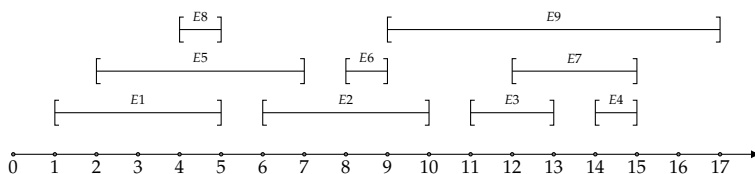
## 13.4 RENDERING A CALENDAR

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

Suppose each day consists of a number of events, where an event is specified as a start time and a finish time. Individual events for a day are to be rendered as non-overlapping rectangular regions whose sides are parallel to the  $x$ - and  $y$ -axes. Let the  $x$ -axis correspond to time. If an event starts at time  $b$  and ends at time  $e$ , the upper and lower sides of its corresponding rectangle must be at  $b$  and  $e$ , respectively. Figure 13.1 on the next 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.

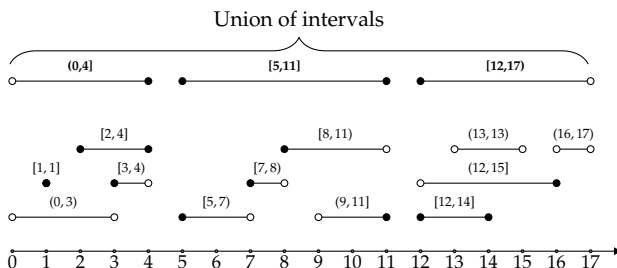


**Figure 13.1:** A set of nine events. The earliest starting event begins at time 1; the latest ending event ends at time 17. The maximum number of concurrent events is 3, e.g.,  $\{E1, E5, E8\}$  as well as others.

**Problem 13.4:** Given a set of events, how would you determine the maximum number of events that take place concurrently? pg. 155

### 13.5 UNION OF INTERVALS

In this problem we consider sets of intervals with integer endpoints; the intervals may be open or closed at either end. We want to compute the union of the intervals in such sets. A concrete example is given in Figure 13.2.



**Figure 13.2:** A set of intervals and their union.

**Problem 13.5:** Design an algorithm that takes as input a set of intervals  $I$ , and outputs the union of the intervals. What is the time complexity of your algorithm as a function of the number of intervals? pg. 156

### 13.6 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  $i_0, i_1, \dots, i_{k-1}$  (not necessarily distinct) such that  $\sum_{j=0}^{k-1} A[i_j] = t$ .

**Problem 13.6:** Design an algorithm that takes as input an array  $A$  and a number  $t$ , and determines if  $A$  3-creates  $t$ . pg. 158

## Binary Search Trees

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

—“A Theorem on Trees,”  
A. CAYLEY, 1889

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

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

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

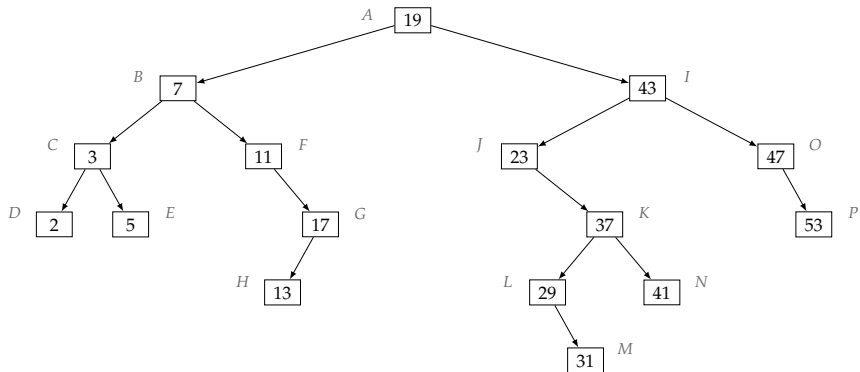
The BST prototype in C++ is listed as follows:

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

### 14.1 DOES A BINARY TREE SATISFY THE BST PROPERTY?

**Problem 14.1:** Write a function that takes as input the root of a binary tree whose nodes have a key field, and returns `true` iff the tree satisfies the BST property. *pg. 159*



**Figure 14.1:** An example BST.

#### 14.2 SEARCH BST FOR THE FIRST KEY LARGER THAN $k$

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

**Problem 14.2:** 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 walk. If  $k$  is absent or no key larger than  $k$  is present, return null. For example, when applied to the BST in Figure 14.1 you should return 29 if  $k = 23$ ; if  $k = 32$ , you should return null. *pg. 162*

#### 14.3 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 14.3:** How would you build a BST of minimum possible height from a sorted array  $A$ ? *pg. 162*

## Meta-algorithms

*The important fact to observe is that we have attempted to solve a maximization problem involving a particular value of  $x$  and a particular value of  $N$  by first solving the general problem involving an arbitrary value of  $x$  and an arbitrary value of  $N$ .*

—“Dynamic Programming,”  
R. E. BELLMAN, 1957

We now cover three general techniques for algorithm design—*divide and conquer*, *dynamic programming*, and the *greedy method*. The approaches described previously, such as mapping a problem into an appropriate data structure, or presorting the input, are more widely used than the methods in this chapter. However, although they are specialized, the approaches in this chapter lead to huge efficiency gains compared to naïve algorithms. These techniques are not exhaustive. In later chapters we will discuss algorithms that use randomization, parallelization, backtracking, heuristic search, reduction, and approximation.

### *Divide and conquer*

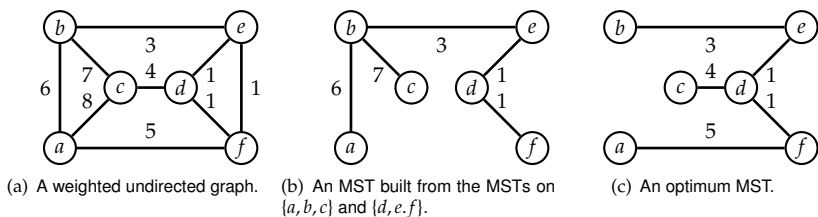
A divide and conquer algorithm works by repeatedly decomposing a problem into two or more smaller independent subproblems of the same kind, until it gets to instances that are simple enough to be solved directly. The solutions to the subproblems are then combined to give a solution to the original problem.

Merge sort and quicksort are classical examples of divide and conquer. In merge sort, the array  $A[0 : n - 1]$  is sorted by sorting  $A[0 : \lfloor n/2 \rfloor]$  and  $A[\lfloor n/2 \rfloor + 1 : n - 1]$ , and merging them. In quicksort,  $A[0 : n - 1]$  is sorted by selecting a pivot element  $A[r]$  and reordering the elements of  $A$  to make all elements appearing before  $A[r]$  less than or equal to  $A[r]$  and all elements appearing after  $A[r]$  greater than or equal to  $A[r]$ . The subarray consisting of elements before  $A[r]$  and the subarray consisting of elements after  $A[r]$  are sorted, and the resulting array is completely sorted.

Interestingly, the divide step in merge sort is trivial; the challenge is in combining the results. With quicksort, the opposite is true. Problems [10.1 on Page 63](#) and [6.1 on Page 49](#) 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 15.1 shows how this algorithm can lead to suboptimal results.

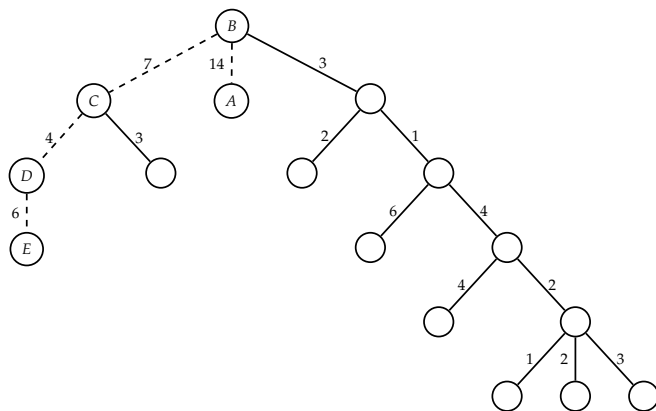


**Figure 15.1:** Divide and conquer applied to the MST problem is suboptimum.

The term divide and conquer is also sometimes applied to algorithms that reduce a problem to only one subproblem, e.g., binary search. Such algorithms can be implemented more efficiently than general divide and conquer algorithms. In particular, these algorithms use tail recursion, which can be replaced by a loop. Decrease and conquer is a more appropriate term for such algorithms.

### 15.1 TREE DIAMETER

Packets in Ethernet local area networks (LANs) are routed according to the unique path in a tree whose nodes correspond to clients and edges correspond to physical connections between the clients. In this problem, we want to design an algorithm for finding the “worst-case” route, i.e., the two clients that are furthest apart. In the abstract, we want to solve the following problem:



**Figure 15.2:** The diameter for the above tree is 31. The corresponding path is  $\langle A, B, C, D, E \rangle$ , which is depicted by the dashed edges.

Let  $T$  be a tree, where each edge is labeled with a nonnegative real-valued distance. Define the diameter of  $T$  to be the length of a longest path in  $T$ . Figure 15.2 illustrates the diameter concept.

**Problem 15.1:** Design an efficient algorithm to compute the diameter of a tree.  
pg. 163

### Dynamic programming

DP is a general technique for solving complex optimization problems that can be decomposed into overlapping subproblems. Like divide and conquer, we solve the problem by combining the solutions of multiple smaller problems but what makes DP different is that the subproblems may not be independent. A key to making DP efficient is reusing the results of intermediate computations. (The word “programming” in dynamic programming does not refer to computer programming—the word was chosen by Richard Bellman to describe a program in the sense of a schedule.) Problems which are naturally solved using DP are a popular choice for hard interview questions.

To illustrate the idea underlying DP, consider the problem of computing Fibonacci numbers defined by  $F_n = F_{n-1} + F_{n-2}$ ,  $F_0 = 0$  and  $F_1 = 1$ . A function to compute  $F_n$  that recursively invokes itself to compute  $F_{n-1}$  and  $F_{n-2}$  would have a time complexity that is exponential in  $n$ . However if we make the observation that recursion leads to computing  $F_i$  for  $i \in [0, n-1]$  repeatedly, we can save the computation time by storing these results and reusing them. This makes the time complexity linear in  $n$ , albeit at the expense of  $O(n)$  storage. Note that the recursive implementation requires  $O(n)$  storage too, though on the stack rather than the heap and that the function is not tail recursive since the last operation performed is  $+$  and not a recursive call.

The key to solving any DP problem efficiently is finding the right way to break the problem into subproblems such that

- the bigger problem can be solved relatively easily once solutions to all the subproblems are available, and
- you need to solve as few subproblems as possible.

In some cases, this may require solving a slightly different optimization problem than the original problem. For example, consider the following problem: given an array of integers  $A$  of length  $n$ , find the interval indices  $a$  and  $b$  such that  $\sum_{i=a}^b A[i]$  is maximized. As a concrete example, the interval corresponding to the maximum subarray sum for the array in Figure 15.3 is  $[0, 3]$ .

904	40	523	12	-335	-385	-124	481	-31
$A[0]$	$A[1]$	$A[2]$	$A[3]$	$A[4]$	$A[5]$	$A[6]$	$A[7]$	$A[8]$

**Figure 15.3:** An array with a maximum subarray sum of 1479.

The brute-force algorithm, which computes each subarray sum, has  $O(n^3)$  time complexity—there are  $\frac{n(n-1)}{2}$  subarrays, and each subarray sum can be computed in  $O(n)$  time. The brute-force algorithm can be improved to  $O(n^2)$  by first computing sums  $S[i]$  for subarrays  $A[0 : i]$  for each  $i < n$ ; the sum of subarray  $A[i : j]$  is  $S[j] - S[i - 1]$ , where  $S[-1]$  is taken to be 0.

Here is a natural divide and conquer algorithm. We solve the problem for the subarrays  $L = A[0 : \lfloor \frac{n}{2} \rfloor]$  and  $R = A[\lfloor \frac{n}{2} \rfloor + 1 : n - 1]$ . In addition to the answers for each, we also return the maximum subarray sum for any subarray ending at  $|L| - 1$  for  $L$  (call this value  $l$ ) and starting at 0 for  $R$  (call this value  $r$ ). The maximum subarray sum for  $A$  is the maximum of  $l + r$ , the answer for  $L$ , and the answer for  $R$ . The time complexity analysis is similar to that for quicksort, which leads to a  $O(n \log n)$ .

Now we will solve this problem by using DP. A natural thought is to assume we have the solution for the subarray  $A[0 : n - 2]$ . However, even if we knew the largest sum subarray for subarray  $A[0 : n - 2]$ , it does not help us solve the problem for  $A[0 : n - 1]$ . A better approach is to iterate through the array. For each index  $j$ , the maximum subarray ending at  $j$  is equal to  $S[j] - \min_{i \leq j} S[i]$ . 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  $\Theta(n)$  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.

15.2 LONGEST NONDECREASING SUBSEQUENCE (🧠)

The problem of finding the longest nondecreasing subsequence in a sequence of integers has implications to many disciplines, including string matching and analyzing card games. As a concrete instance, the length of a longest nondecreasing subsequence for the array  $A$  in Figure 15.4 is 4. There are multiple longest nondecreasing subsequences, e.g.,  $\langle 0, 4, 10, 14 \rangle$  and  $\langle 0, 2, 6, 9 \rangle$ .

0	8	4	12	2	10	6	14	1	9
$A[0]$	$A[1]$	$A[2]$	$A[3]$	$A[4]$	$A[5]$	$A[6]$	$A[7]$	$A[8]$	$A[9]$

Figure 15.4: An array whose longest nondecreasing subsequences are of length 4.

**Problem 15.2:** Given an array  $A$  of  $n$  numbers, find a longest subsequence  $\langle i_0, \dots, i_{k-1} \rangle$  such that  $i_j < i_{j+1}$  and  $A[i_j] \leq A[i_{j+1}]$  for any  $j \in [0, k - 2]$ . pg. 164

15.3 LEVENSHTAIN DISTANCES

Spell checkers make suggestions for misspelled words. Given a misspelled string  $s$ , a spell checker should return words in the dictionary which are close to  $s$ .

In 1965, Vladimir Levenshtein defined the distance between two words as the minimum number of “edits” it would take to transform the misspelled word into a correct word, where a single edit is the *insertion*, *deletion*, or *substitution* of a single character.

**Problem 15.3:** Given two strings, represented as arrays of characters  $A$  and  $B$ , compute the minimum number of edits needed to transform the first string into the second string. pg. 166

15.4 WORD BREAKING

Suppose you are designing a search engine. In addition to getting keywords from a page’s content, you would like to get keywords from Uniform Resource Locators (URLs). For example, `bedbathandbeyond.com` should be associated with “bed bath and beyond” (in this version of the problem we also allow “bed bat hand beyond” to be associated with it).

**Problem 15.4:** Given a dictionary and a string  $s$ , design an efficient algorithm that checks whether  $s$  is the concatenation of a sequence of dictionary words. If such a concatenation exists, your algorithm should output it. pg. 168

15.5 SCORE COMBINATIONS

In an American football game, a play can lead to 2 points (safety), 3 points (field goal), or 7 points (touchdown). Given the final score of a game, we want to compute how many different combinations of 2, 3, and 7 point plays could make up this score.

For example, if  $W = \{2, 3, 7\}$ , four combinations of plays yield a score of 12:

- 6 safeties ( $2 \times 6 = 12$ ),
- 3 safeties and 2 field goals ( $2 \times 3 + 3 \times 2 = 12$ ),
- 1 safety, 1 field goal and 1 touchdown ( $2 \times 1 + 3 \times 1 + 7 \times 1 = 12$ ), and
- 4 field goals ( $3 \times 4 = 12$ ).

**Problem 15.5:** 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. 169

## 15.6 NUMBER OF WAYS

Suppose you start at the top-left corner of an  $n \times m$  2D array  $A$  and want to get to the bottom-right corner. The only way you can move is by either going right or going down. Three legal paths for a  $5 \times 5$  2D array are given in Figure 15.5.

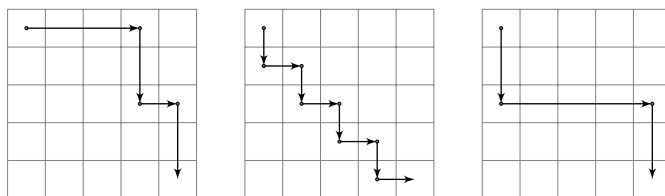


Figure 15.5: Paths through a 2D array.

**Problem 15.6:** 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. 170

### The greedy method

As described on Page 31, the greedy method is an algorithm design pattern which results in an algorithm that computes a solution in steps. At each step the algorithm makes a decision that is locally optimum, and never changes that decision.

The example on Page 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 (Chapter 17) but for some coinages, the greedy algorithm is optimum—e.g., if the denominations are of the form  $\{1, r, r^2, r^3\}$ . (An *ad hoc* argument can be applied to show that the greedy

algorithm is also optimum for US coinage.) The general problem can be solved in pseudo-polynomial time using DP in a manner similar to Problem 17.2 on Page 92.

15.7 HUFFMAN CODING (🔒)

One way to compress a large text is by building a code book which maps each character to a bit string, referred to as its code word. Compression consists of concatenating the bit strings for each character to form a bit string for the entire text.

When decompressing the string, we read bits until we find a string that is in the code book and then repeat this process until the entire text is decoded. For the compression to be reversible, it is sufficient that the code words have the property that no code word is a prefix of another. For example, 011 is a prefix of 0110 but not a prefix of 1100.

Since our objective is to compress the text, we would like to assign the shorter strings to more common characters and the longer strings to less common characters. We will restrict our attention to individual characters. (We may achieve better compression if we examine common sequences of characters, but this increases the time complexity.)

The intuitive notion of commonness is formalized by the *frequency* of a character which is a number between zero and one. The sum of the frequencies of all the characters is 1. The average code length is defined to be the sum of the product of the length of each character’s code word with that character’s frequency. Table 15.1 shows the large variation in the frequencies of letters of the English alphabet.

**Table 15.1:** English characters and their frequencies, expressed as percentages, in everyday documents.

Character	Frequency	Character	Frequency	Character	Frequency
a	8.17	j	0.15	s	6.33
b	1.49	k	0.77	t	9.06
c	2.78	l	4.03	u	2.76
d	4.25	m	2.41	v	0.98
e	12.70	n	6.75	w	2.36
f	2.23	o	7.51	x	0.15
g	2.02	p	1.93	y	1.97
h	6.09	q	0.10	z	0.07
i	6.97	r	5.99		

**Problem 15.7:** Given a set of symbols with corresponding frequencies, find a code book that has the smallest average code length. pg. 172



# Graphs

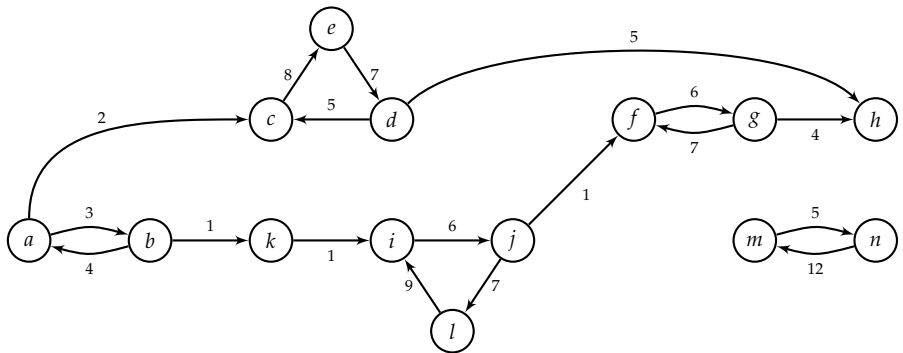
Concerning these bridges, it was asked whether anyone could arrange a route in such a way that he would cross each bridge once and only once.

— “The solution of a problem relating to the geometry of position,”  
L. EULER, 1741

Informally, a graph is a set of vertices and connected by edges. Formally, a directed graph is a tuple  $(V, E)$ , where  $V$  is a set of *vertices* and  $E \subset V \times V$  is the set of edges. Given an edge  $e = (u, v)$ , the vertex  $u$  is its *source*, and  $v$  is its *sink*. Graphs are often decorated, e.g., by adding lengths to edges, weights to vertices, and a start vertex. A directed graph can be depicted pictorially as in Figure 16.1.

A *path* in a directed graph from  $u$  to vertex  $v$  is a sequence of vertices  $\langle v_0, v_1, \dots, v_{n-1} \rangle$  where  $v_0 = u$ ,  $v_{n-1} = v$ , and  $(v_i, v_{i+1}) \in E$  for  $i \in \{0, \dots, n-2\}$ . The sequence may contain a single vertex. The *length* of the path  $\langle v_0, v_1, \dots, v_{n-1} \rangle$  is  $n-1$ . Intuitively, the *length* of a path is the number of edges it traverses. If there exists a path from  $u$  to  $v$ ,  $v$  is said to be *reachable* from  $u$ .

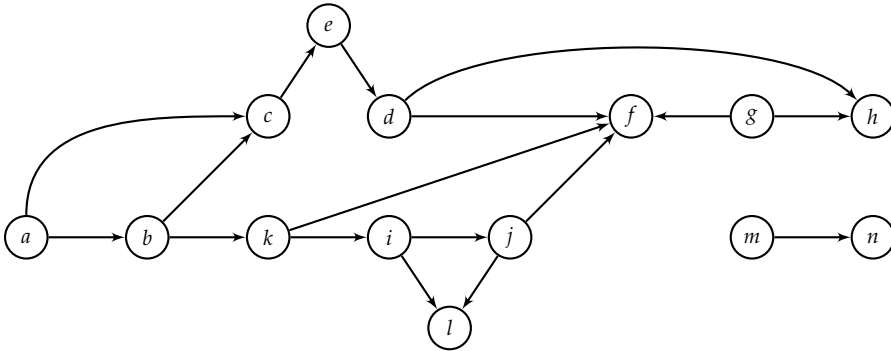
For example, the sequence  $\langle a, c, e, d, h \rangle$  is a path in the graph represented in Figure 16.1.



**Figure 16.1:** A directed graph with weights on edges.

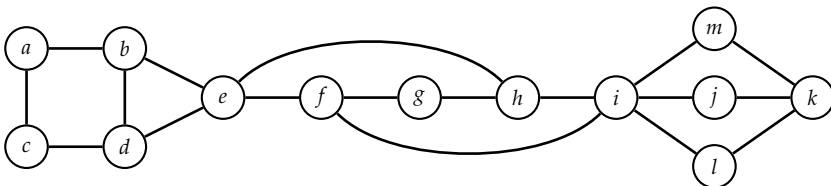
A *directed acyclic graph* (DAG) is a directed graph in which there are no *cycles*, i.e., paths of the form  $\langle v_0, v_1, \dots, v_{n-1}, v_0 \rangle$ ,  $n \geq 1$ . See Figure 16.2 on the following page for an example of a directed acyclic graph. Vertices in a DAG which have no incoming

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. Solution 16.3 on Page 178 uses the notion of topological ordering.



**Figure 16.2:** 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 16.3.



**Figure 16.3:** 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 9, differ from ordered trees since a node may have only one child in a binary tree, but that node may be a left or a right child, whereas in an ordered tree no analogous notion exists for a node with a single child. Specifically, in a binary tree, there is position as well as order associated with the children of nodes.

As an example, the graph in Figure 16.4 is a tree. Note that its edge set is a subset of the edge set of the undirected graph in Figure 16.3 on the facing 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  $G$ .

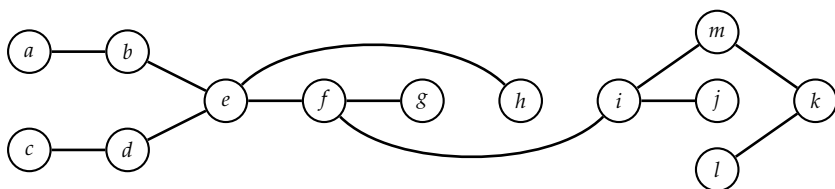


Figure 16.4: A tree.

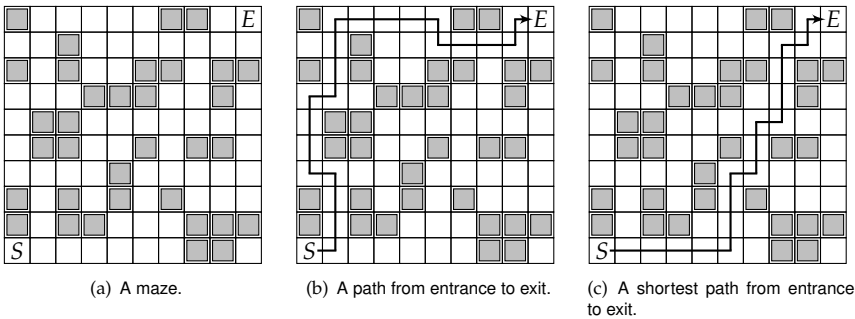
## Graph search

Computing vertices which are reachable from other vertices is a fundamental operation which can be performed by depth-first search (DFS) and breadth-first search (BFS). Both are linear time— $O(|V| + |E|)$ . They differ from each other in terms of the additional information they provide, e.g., BFS can be used to compute distances from the start vertex and DFS can be used to check for the presence of cycles. Key notions in DFS include the concept of *discovery time* and *finishing time* for vertices.

### 16.1 SEARCHING A MAZE

It is natural to apply graph models and algorithms to spatial problems. Consider a black and white digitized image of a maze—white pixels represent open areas and black spaces are walls. There are two special white pixels: one is designated the

entrance and the other is the exit. The goal in this problem is to find a way of getting from the entrance to the exit, as illustrated in Figure 16.5.



**Figure 16.5:** An instance of the maze search problem, with two solutions, where *S* and *E* denote the entrance and exit, respectively.

**Problem 16.1:** Given a 2D array of black and white entries representing a maze with designated entrance and exit points, find a path from the entrance to the exit, if one exists. pg. 175

## 16.2 TRANSFORM ONE STRING TO ANOTHER (🧠)

Let *s* and *t* be strings and *D* a dictionary, i.e., a set of strings. Define *s* to *produce* *t* if there exists a sequence of strings  $\sigma = \langle s_0, s_1, \dots, s_{n-1} \rangle$  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  $\sigma$  is called a *production sequence*.

**Problem 16.2:** Given a dictionary *D* and two strings *s* and *t*, write a function to determine if *s* produces *t*. Assume that all characters are lowercase alphabets. If *s* does produce *t*, output the length of a shortest production sequence; otherwise, output -1. pg. 177

### 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 and minimum spanning tree problems: these are subjects which anyone interviewing for a software position should be familiar with.

### 16.3 TEAM PHOTO DAY—2

**Problem 16.3:** You are the photographer at a sporting event. You have to take pictures of teams. Each team has the same number of players. A team photo consist of rows of players. Each row consists of players from from one team. Each player must be taller than the player in front of him. Players within a row are equally spaced. pg. 178

### 16.4 MINIMUM DELAY SCHEDULE, UNLIMITED RESOURCES

Let  $\mathcal{T} = \{T_0, T_1, \dots, T_{n-1}\}$  be a set of tasks. Each task runs on a single generic server. Task  $T_i$  has a duration  $\tau_i$ , 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  $\langle T_0, T_1, \dots, T_{n-1}, T_0 \rangle$  starting and ending at the same task such that for each consecutive pair of tasks in the sequence, the first task must be completed before the second task can begin.

**Problem 16.4:** 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. 179

### 16.5 SHORTEST PATH WITH FEWEST EDGES

In the usual formulation of the shortest path problem, the number of edges in the path is not a consideration. For example, considering the shortest path problem from  $a$  to  $h$  in Figure 16.1 on Page 85, the sum of the edge costs on the path  $\langle a, c, e, d, h \rangle$  is

22, which is the same as for path  $\langle a, b, k, i, j, f, g, h \rangle$ . Both are shortest paths, but the latter has three more edges.

Heuristically, if we did want to avoid paths with a large number of edges, we can add a small amount to the cost of each edge. However depending on the structure of the graph and the edge costs, this may not result in the shortest path.

**Problem 16.5:** Design an algorithm which takes as input a graph  $G = (V, E)$ , directed or undirected, a nonnegative cost function on  $E$ , and vertices  $s$  and  $t$ ; your algorithm should output a path with the fewest edges amongst all shortest paths from  $s$  to  $t$ .

pg. 179

## Intractability

*All of the general methods presently known for computing the chromatic number of a graph, deciding whether a graph has a Hamiltonian cycle, or solving a system of linear inequalities in which the variables are constrained to be 0 or 1, require a combinatorial search for which the worst-case time requirement grows exponentially with the length of the input.*

—“Reducibility Among Combinatorial Problems,”

R. M. KARP, 1972

In real-world settings you will sometimes encounter problems that can be directly solved using efficient textbook algorithms such as binary search and shortest paths. As we have seen in the earlier chapters, it is often difficult to identify such problems because the core algorithmic problem is obscured by details. More generally, you may encounter problems which can be transformed into equivalent problems that have an efficient textbook algorithm, or problems that can be solved efficiently using meta-algorithms such as DP.

Often the problem you are given is intractable—i.e., there may not exist an efficient algorithm for the problem. Complexity theory addresses these problems. Some have been proved to not have an efficient solution (such as checking the validity of relationships involving  $\exists$ ,  $+$ ,  $<$ ,  $\Rightarrow$  on the integers) but the vast majority are only conjectured to be intractable. The conjunctive normal form satisfiability (CNF-SAT) problem 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 which are typically exponential but may be acceptable, if the instances encountered are small.
- Branch and bound techniques which prune much of the complexity of a brute-force search.
- Approximation algorithms which return a solution that is provably close to optimum.

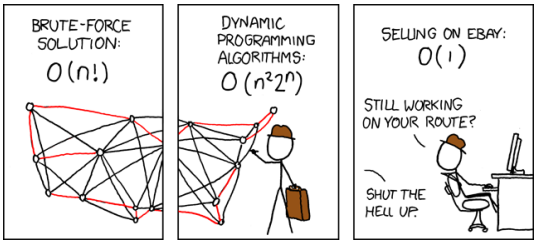


Figure 17.1:  $P = NP$ , by XKCD.

- Heuristics based on insight, common case analysis, and careful tuning that may solve the problem reasonably well.
- Parallel algorithms, wherein a large number of computers can work on subparts simultaneously.

Don't forget it may be possible to dramatically change the problem formulation while still achieving the higher level goal, as illustrated in Figure 17.1.

17.1 TIES IN A PRESIDENTIAL ELECTION

The US President is elected by the members of the Electoral College. The number of electors per state and Washington, D.C., are given in Table 17.1. All electors from each state as well as Washington, D.C., cast their vote for the same candidate.

Table 17.1: Electoral college votes.

State	Electors	State	Electors	State	Electors
Alabama	9	Louisiana	8	Ohio	18
Alaska	3	Maine	4	Oklahoma	7
Arizona	11	Maryland	10	Oregon	7
Arkansas	6	Massachusetts	11	Pennsylvania	20
California	55	Michigan	16	Rhode Island	4
Colorado	9	Minnesota	10	South Carolina	9
Connecticut	7	Mississippi	6	South Dakota	3
Delaware	3	Missouri	10	Tennessee	11
Florida	29	Montana	3	Texas	38
Georgia	16	Nebraska	5	Utah	6
Hawaii	4	Nevada	6	Vermont	3
Idaho	4	New Hampshire	4	Virginia	13
Illinois	20	New Jersey	14	Washington	12
Indiana	11	New Mexico	5	West Virginia	5
Iowa	6	New York	29	Wisconsin	10
Kansas	6	North Carolina	15	Wyoming	3
Kentucky	8	North Dakota	3	Washington, D.C.	3

**Problem 17.1:** How would you programmatically determine if a tie is possible in a presidential election with two candidates,  $R$  and  $D$ ? pg. 181

17.2 THE KNAPSACK PROBLEM

A thief breaks into a clock store. His knapsack will hold at most  $w$  ounces of clocks. Clock  $i$  weighs  $w_i$  ounces and retails for  $v_i$  dollars. The thief must either take or leave



a clock, and he cannot take a fractional amount of an item. His intention is to take clocks whose total value is maximum subject to the knapsack capacity constraint. His problem is illustrated in Figure 17.2. If the knapsack can hold at most 130 ounces, he cannot take all the clocks. If he greedily chooses clocks, in decreasing order of value-to-weight ratio, he will choose  $P, H, O, B, I$ , and  $L$  in that order for a total value of \$669. However,  $\{H, J, O\}$  is the optimum selection, yielding a total value of \$695.

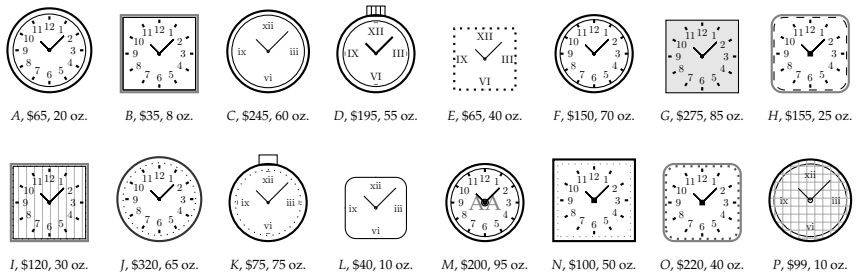


Figure 17.2: A clock store.

**Problem 17.2:** Design an algorithm for the knapsack problem that selects a subset of items that has maximum value and weighs at most  $w$  ounces. All items have integer weights and values. pg. 181

### 17.3 MEASURING WITH DEFECTIVE JUGS

You have three measuring jugs,  $A$ ,  $B$ , and  $C$ . The measuring marks have worn out, making it impossible to measure exact volumes. Specifically, each time you measure with  $A$ , all you can be sure of is that you have a volume that is in the range  $[230, 240]$  mL. (The next time you use  $A$ , you may get a different volume—all that you know with certainty is that the quantity will be in  $[230, 240]$  mL.) Jugs  $B$  and  $C$  can be used to measure a volume in  $[290, 310]$  mL and in  $[500, 515]$  mL, respectively. Your recipe for chocolate chip cookies calls for at least 2100 mL and no more than 2300 mL of milk.

**Problem 17.3:** Write a program that determines a sequence of steps by which the required amount of milk can be obtained using the worn-out jugs. The milk is being added to a large mixing bowl, and hence cannot be removed from the bowl. Furthermore, it is not possible to pour one jug's contents into another. Your scheme should always work, i.e., return between 2100 and 2300 mL of milk, independent of how much is chosen in each individual step, as long as that quantity satisfies the given constraints. pg. 182

### 17.4 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  $9 \times 9$  grid.

**Problem 17.4:** 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. 183*

---

## Parallel Computing

*The activity of a computer must include the proper reacting to a possibly great variety of messages that can be sent to it at unpredictable moments, a situation which occurs in all information systems in which a number of computers are coupled to each other.*

---

— “Cooperating sequential processes,”  
E. W. DIJKSTRA, 1965

Parallel computation has become increasingly common. For example, laptops and desktops come with multiple processors which communicate through shared memory. High-end computation is often done using clusters consisting of individual computers communicating through a network.

Parallelism provides a number of benefits:

- High performance—more processors working on a task (usually) means it is completed faster.
- Better use of resources—a program can execute while another waits on the disk or network.
- Fairness—letting different users or programs share a machine rather than have one program run at a time to completion.
- Convenience—it is often conceptually more straightforward to do a task using a set of concurrent programs for the subtasks rather than have a single program manage all the subtasks.
- Fault tolerance—if a machine fails in a cluster that is serving web pages, the others can take over.

Concrete applications of parallel computing include graphical user interfaces (GUI) (a dedicated thread handles UI actions resulting in increased responsiveness), Java virtual machines (a separate thread handles garbage collection which would otherwise lead to blocking), web servers (a single logical thread handles a single client request), scientific computing (a large matrix multiplication can be split across a cluster), and web search (multiple machines crawl, index, and retrieve web pages).

The two primary models for parallel computation are the shared memory model, in which each processor can access any location in memory, and the distributed memory model, in which a processor must explicitly send a message to another processor to access its memory. The former is more appropriate in the multicore setting and the latter is more accurate for a cluster. The questions in this chapter are mostly focused on the shared memory model. We cover a few problems related to

the distributed memory model, such as leader election and sorting large data sets, at the end of the chapter.

Writing correct parallel programs is challenging because of the subtle interactions between parallel components. One of the key challenges is races—two concurrent instruction sequences access the same address in memory and at least one of them writes to that address. Other challenges to correctness are

- starvation (a processor needs a resource but never gets it, e.g., Problem 18.3),
- deadlock (Thread *A* acquires Lock *L1* and Thread *B* acquires Lock *L2*, following which *A* tries to acquire *L2* and *B* tries to acquire *L1*), and
- livelock (a processor keeps retrying an operation that always fails).

Bugs caused by these issues are difficult to find using testing. Debugging them is also difficult because they may not be reproducible since they are usually load dependent. It is also often true that it is not possible to realize the performance implied by parallelism—sometimes a critical task cannot be parallelized, making it impossible to improve performance, regardless of the number of processors added. Similarly, the overhead of communicating intermediate results between processors can exceed the performance benefits.

## 18.1 SERVICE WITH CACHING

**Problem 18.1:** Design an online spell correction system. It should take as input a string *s* and return an array of entries in its dictionary which are closest to the string using the Levenshtein distance specified in Problem 15.3 on Page 82. Cache the most recently computed result. *pg. 185*

## 18.2 TIMER

Consider a web-based calendar in which the server hosting the calendar has to perform a task when the next calendar event takes place. (The task could be sending an email or a Short Message Service (SMS).) Your job is to design a facility that manages the execution of such tasks.

**Problem 18.2:** 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. 187*

## 18.3 READERS-WRITERS

Consider an object *s* which is read from and written to by many threads. (For example, *s* could be the cache from Problem 18.1.) You need to ensure that no thread may access *s* for reading or writing while another thread is writing to *s*. (Two or more readers may access *s* at the same time.)

One way to achieve this is by protecting *s* with a mutex that ensures that two threads cannot access *s* at the same time. However this solution is suboptimal

because it is possible that a reader  $R1$  has locked  $s$  and another reader  $R2$  wants to access  $s$ . Reader  $R2$  does not have to wait until  $R1$  is done reading; instead,  $R2$  should start reading right away.

This motivates the first readers-writers problem: protect  $s$  with the added constraint that no reader is to be kept waiting if  $s$  is currently opened for reading.

**Problem 18.3:** Implement a synchronization mechanism for the first readers-writers problem. *pg. 187*

## Design Problems

*We have described a simple but very powerful and flexible protocol which provides for variation in individual network packet sizes, transmission failures, sequencing, flow control, and the creation and destruction of process- to-process associations.*

— “A Protocol for Packet Network Intercommunication,”  
V. G. CERF AND R. E. KAHN, 1974

This chapter is concerned with system design problems. These problems are fairly open-ended, and many can be the starting point for a large software project for a Ph.D. A comprehensive discussion on the solutions available for such problems is outside the scope of this book. In an interview setting when someone asks such a question, you should have a conversation in which you demonstrate an ability to think creatively, understand design trade-offs, and attack unfamiliar problems. You should sketch key data structures and algorithms, as well as the technology stack (programming language, libraries, OS, hardware, and services) that you would use to solve the problem. Some specific things to keep in mind are implementation time, scalability, testability, security, and internationalization.

The answers in this chapter are presented in this context—they are meant to be examples of good responses in an interview and are not definitive state-of-the-art solutions.

### 19.1 CREATING PHOTOMOSAICS

A photomosaic is built from a collection of images called “tiles” and a target image. The photomosaic is another image which approximates the target image and is built by juxtaposing the tiles. The quality of approximation is defined by human perception.

**Problem 19.1:** Design a program that produces high quality mosaics with minimal compute time. *pg. 188*

### 19.2 IMPLEMENT PAGERANK

The PageRank algorithm assigns a rank to a web page based on the number of “important” pages that link to it. The algorithm essentially amounts to the following:

1. Build a matrix  $A$  based on the hyperlink structure of the web. Specifically,  $A_{ij} = \frac{1}{d_i}$  if page  $i$  links to page  $j$ ; here  $d_i$  is the total number of unique outgoing links from page  $i$ .

2. Solve for  $X$  satisfying  $X = \epsilon[\mathbf{1}] + (1 - \epsilon)A^T X$ . Here  $\epsilon$  is a scalar 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 19.2:** Design a system that can compute the ranks of ten billion web pages in a reasonable amount of time. *pg. 189*

# Probability

*The theory of probability, as a mathematical discipline, can and should be developed from axioms in exactly the same way as Geometry and Algebra.*

— “Foundations of the Theory of Probability,”  
A. N. KOLMOGOROV, 1933

Probability comes up often in computation, e.g., when modeling random events (input data and arrival time), and designing efficient algorithms, quicksort and selecting the  $k$ -th element being notable examples. It is a rich subject and is the source of many interview questions.

To a first approximation, a probability measure is a function  $p$  from subsets of a set  $E$  of events to  $[0, 1]$  that has the properties that  $p(E) = 1$  and  $p(A \cup B) = p(A) + p(B)$  when  $A$  and  $B$  are disjoint. Various properties and notations can be given around these concepts. For example, it is easy to prove that  $p(A \cup B) = p(A) + p(B) - p(A \cap B)$  for general  $A$  and  $B$ .

A random variable  $X$  is a function from  $E$  to  $(-\infty, \infty)$ ; it can be characterized by a *cumulative distribution function*  $F_X : \mathfrak{R} \mapsto [0, 1]$ , where  $F_X(\tau) = p(X^{-1}((-\infty, \tau]))$ . When  $X$  takes a finite or countable set of values, we can talk about the probability of  $X$  taking a particular value, i.e.,  $p(X = \tau_i)$ . If  $X$  takes a continuous range of values and  $F_X$  is differentiable, we talk of  $f_X(\tau) = \frac{dF_X}{d\tau}$  as being the *probability density function*.

The expected value  $E[X]$  of a random variable  $X$  taking a finite set of values  $T = \{\tau_0, \tau_1, \dots, \tau_{n-1}\}$  is simply  $\sum_{\tau_i \in T} \tau_i \cdot p(X = \tau_i)$ , i.e., it is the average value of  $X$ , weighted by probabilities. The notion of expectation generalizes to countable sets of values. For a random variable taking a continuous set of values, the sum is replaced with an integral and the weighting function is the probability density function. The variance  $\text{var}(X)$  of a random variable  $X$  is the expected value of  $(X - E[X])^2$ , and, in some sense, measures how spread out the variable is. Some of the key results in probability have to do with bounds on the probability of events, e.g., the Chebyshev bound:  $\Pr(|X - E[X]| \geq k \sqrt{\text{var}(X)}) \leq \frac{1}{k^2}$  holds for arbitrary distributions.

The following random variables are frequently encountered. The Bernoulli random variable takes only two values, 0 and 1; it is used, for example, in modeling coin tosses. The uniform random variable takes values in an interval  $U$ ; the probability of  $I \subset U$  is proportional to the length of  $I$ . The Poisson random variable takes non-negative values—it models the number of events in a fixed period of time, e.g., the number of HTTP requests in a minute. The Gaussian random variable takes all real values. Let  $X_0, X_1, X_2, \dots$  be independent identically distributed random variables



each with mean  $\mu$  and variance  $\sigma^2$ . Then  $(\sum_{i=0}^{n-1} (X_i - \mu)) / \sqrt{n}$  tends to a zero mean Gaussian random variable with unit variance.

For the most part, probability is intuitive. However, there are notable exceptions. For example, at first glance, it would seem impossible for there to exist three 6-sided dice  $A$ ,  $B$ , and  $C$  such that  $A$  is more likely to roll a higher number than  $B$ ,  $B$  is more likely to roll a higher number than  $C$ , and  $C$  is more likely to roll a higher number than  $A$ . However if  $A$  has sides 2, 2, 4, 4, 9, and 9,  $B$  has sides 1, 1, 6, 6, 8, and 8, and die  $C$  has sides 3, 3, 5, 5, 7, and 7, then the probability that  $A$  rolls a higher number than  $B$  is  $\frac{20}{36}$ , the probability that  $B$  rolls a higher number than  $C$  is  $\frac{20}{36}$ , and the probability that  $C$  rolls a higher number than  $A$  is  $\frac{20}{36}$ . The Monty Hall problem is another famous example.

## 20.1 OFFLINE SAMPLING

**Problem 20.1:** Let  $A$  be an array of  $n$  distinct elements. Design an algorithm that returns a subset of  $k$  elements of  $A$ . All subsets should be equally likely. Use as few calls to the random number generator as possible and use  $O(1)$  additional storage. You can return the result in the same array as input. pg. 190

## 20.2 UNIFORM RANDOM NUMBER GENERATION

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

**Problem 20.2:** How would you implement a random number generator that generates a random integer  $i$  in  $[a, b]$ , given a random number generator that produces either zero or one with equal probability? All generated values should have equal probability. What is the run time of your algorithm, assuming each call to the given random number generator takes  $O(1)$  time? pg. 191

## 20.3 RESERVOIR SAMPLING

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

**Problem 20.3:** Design an algorithm that reads a sequence of packets and maintains a uniform random subset of size  $k$  of the read packets when the  $n \geq k$ -th packet is read. pg. 192

## 20.4 ONLINE SAMPLING

The set  $\mathcal{Z}_n = \{0, 1, 2, \dots, n-1\}$  has exactly  $\binom{n}{k}$  subsets of size  $k$ . We seek to design an algorithm that returns any one of these subsets with equal probability.

**Problem 20.4:** Design an algorithm that computes an array of size  $k$  consisting of distinct integers in the set  $\{0, 1, \dots, n-1\}$ . All subsets should be equally likely and,

in addition, all permutations of elements of the array should be equally likely. Your time should be  $O(k)$ . Your algorithm should use  $O(k)$  space in addition to the  $k$  element array holding the result. You may assume the existence of a subroutine that returns integers in the set  $\{0, 1, \dots, n - 1\}$  with uniform probability. pg. 193

## Discrete Mathematics

*There is required, finally, the ratio between the fluxion of any quantity  $x$  you will and the fluxion of its power  $x^n$ . Let  $x$  flow till it becomes  $x + o$  and resolve the power  $(x + o)^n$  into the infinite series  $x^n + nox^{n-1} + \frac{1}{2}(n^2 - n)o^2x^{n-2} + \frac{1}{6}(n^3 - 3n^2 + 2n)o^3x^{n-3} \dots$*

—“On the Quadrature of Curves,”

I. NEWTON, 1693

Discrete mathematics is used in algorithm design in a variety of places. Examples include combinatorial optimization, complexity analysis, and bounding probabilities. Discrete mathematics is also the source of enjoyable puzzles and challenging interview questions. Solutions can range from simple applications of the pigeon-hole principle to complex inductive reasoning.

Some of the problems in this chapter fall into the category of brain teasers where all that is needed is an *aha* moment. These problems have fallen out of fashion because it is difficult to judge a candidate’s ability based on whether he is able to make a tricky observation in a short period of time. However they are asked often enough that it is important to understand basic principles.

### 21.1 500 DOORS

Five hundred closed doors along a corridor are numbered from 1 to 500. A person walks through the corridor and opens each door. Another person walks through the corridor and closes every alternate door. Continuing in this manner, the  $i$ -th person comes and toggles the position of every  $i$ -th door starting from door  $i$ .

**Problem 21.1:** Which of the 500 doors are open after the 500-th person has walked through? pg. 194

### 21.2 EFFICIENT TRIALS

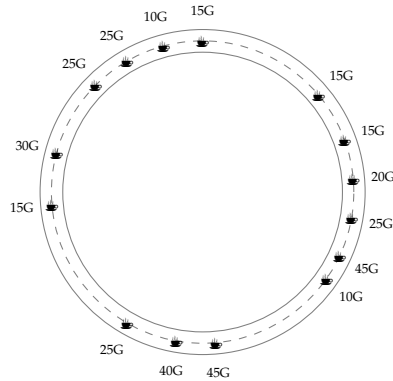
You are the coach of a cycling team with 25 members and need to determine the fastest, second-fastest, and third-fastest cyclists for selection to the Olympic team.

You will be evaluating the cyclists using a time-trial course on which only five cyclists can race at a time. You can use the completion times from a time-trial to rank the five cyclists amongst themselves—no ties are possible. Because conditions can change over time, you cannot compare performances across different time-trials. The

relative speeds of cyclists does not change—if  $a$  beats  $b$  in one time-trial and  $b$  beats  $c$  in another time-trial, then  $a$  is guaranteed to beat  $c$  if they are in the same time-trial.

**Problem 21.2:** What is the minimum number of five man time-trials needed to determine the top three cyclists from a set of 25 cyclists? pg. 195

### 21.3 THE GASUP PROBLEM



**Figure 21.1:** The length of the circular route is 7200 miles, and your vehicle gets 20 miles per gallon. The distance between successive gas stations is proportional to the angle they subtend at the center.

In the gasup problem,  $n$  cities are arranged on a circular road. You need to visit all the  $n$  cities and come back to the starting city. A certain amount of gas is available at each city. The total amount of gas is equal to the amount of gas required to go around the road once. Your gas tank has unlimited capacity. Call a city ample if you can begin at it with an empty tank, travel through all the remaining cities, refilling at each, and return to it. An instance of this problem is given in Figure 21.1.

**Problem 21.3:** Given an instance of the gasup problem, how would you efficiently compute an ample city if one exists? pg. 195

### 21.4 THE MAXIMUM PRODUCT OF $(n - 1)$ NUMBERS (🧠)

You are given an array  $A$  of  $n$  elements,  $n \geq 2$ , and are asked to find the  $n - 1$  elements in  $A$  which have the largest product.

One approach is to form the product  $P = \prod_{i=0}^{n-1} A[i]$ , and then find the maximum of the  $n$  terms  $P_i = P/A[i]$ ; this takes  $n - 1$  multiplications and  $n$  divisions. Suppose because of finite precision considerations we cannot use the division-based approach described above; we can only use multiplications. The brute-force solution entails computing all  $\binom{n}{n-1} = n$  products of  $n - 1$  elements; each such product takes  $n - 2$  multiplications.

**Problem 21.4:** Given an array  $A$  with  $n$  elements, compute  $\max_{j=0}^{n-1} \frac{\prod_{i=0}^{n-1} A[i]}{A[j]}$  in  $O(n)$  time without using division. Can you design an algorithm that runs in  $O(1)$  space and  $O(n)$  time? Array entries may be positive, negative, or 0. pg. 196

## 21.5 HEIGHT DETERMINATION (🔒)

You need to test the design of a protective case. Specifically, the case can protect the enclosed device from a fall from up to some number of floors, and you want to determine what that number of floors is. You want to achieve this using no more than  $c$  cases. An additional constraint is that you can perform only  $d$  drops before the building supervisor stops you.

You know that there exists a floor  $x$  such that the case will break if it is dropped from any floor  $x$  or higher but will remain intact if dropped from a floor below  $x$ . The ground floor is numbered zero, and it is given that the case will not break if dropped from the ground floor.

**Problem 21.5:** Given  $c$  cases and  $d$  drops, what is the maximum number of floors that you can test in the worst-case? pg. 198

## 21.6 DANCING WITH THE STARS (🔒)

You are organizing a celebrity dance for charity. Specifically, a number of celebrities have offered to be partners for a ballroom dance. The general public has been invited to offer bids on how much they are willing to pay for a dance with each celebrity.

**Problem 21.6:** Design an algorithm for pairing bidders with celebrities to maximize the revenue from the dance. Each celebrity cannot dance more than once, and each bidder cannot dance more than once. Assume that the set of celebrities is disjoint from the set of bidders. How would you modify your approach if all bids were for the same amount? What if celebrities and bidders are not disjoint? pg. 198

Part III

Hints

---

## Hints

*When I interview people, and they give me an immediate answer, they're often not thinking. So I'm silent. I wait. Because they think they have to keep answering. And it's the second train of thought that's the better answer.*

---

— R. LEACH

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: There are  $2^{|S|}$  subsets for a given set  $S$ . There are  $2^k$   $k$ -bit words.

5.3: Build the result one digit at a time.

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

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

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

6.5: The input definition is recursive.

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

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

7.3: Solve the simple cases first.

7.4: Use a pair of pointers.

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

8.1: Use additional storage to track the maximum value.

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

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

- 9.1: The definition of symmetry is recursive.
- 9.2: How can you tell whether a node is a left child or right child of its parent?
- 9.3: Study  $n$ 's right subtree. What if  $n$  does not have a right subtree?
- 9.4: When is the root the LCA?
- 10.1: Which portion of each file is significant as the algorithm executes?
- 10.2: Can you cast this in terms of combining  $k$  sorted arrays?
- 10.3: 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?
- 10.4: How many numbers must you read after reading the  $i$ -th number  $x_i$  to be sure you can place  $x_i$  in the correct location?
- 10.5: Systematically enumerate points.
- 11.1: Don't stop after you reach the first  $k$ . Think about the case where every entry equals  $k$ .
- 11.2: This problem easy to solve with  $O(n)$  additional space—why? To solve it with  $O(1)$  additional space, first assume all elements are positive.
- 11.3: Use the decrease and conquer principle.
- 11.4: 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.
- 11.5: Iteratively compute a sequence of intervals, each contained in the previous interval, that contain the result.
- 11.6: Can you be sure there is an address which is not in the file?
- 11.7: Take advantage of the existence of a majority element to perform elimination.
- 12.1: Map strings to strings so that strings which are anagrams map to the same string.
- 12.2: Count.
- 12.3: A line can be uniquely represented by two numbers.
- 13.1: Add a degree of indirection.
- 13.2: Partition the array into subarrays which hold objects with equal keys.
- 13.3: Solve the problem if  $n$  and  $m$  differ by orders of magnitude. What if  $n \approx m$ ?
- 13.4: Focus on endpoints.
- 13.5: Do a case analysis.
- 13.6: How would you check if  $A[i]$  is part of a triple that 3-creates  $t$ ?
- 14.1: 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?
- 14.2: Perform binary search, keeping some additional state.
- 14.3: Which element should be the root?
- 15.1: The longest path may or may not pass through the root.
- 15.2: Express the longest nondecreasing subsequence ending at  $A[i]$  in terms of the longest nondecreasing subsequence in  $A[0 : i - 1]$ .



- 15.3: Consider the same problem for  $A[0 : i - 1]$  and  $B[0 : j - 1]$ .
- 15.4: Solve the generalized problem, i.e., determine for each prefix of  $s$  whether it is the concatenation of dictionary words.
- 15.5: Count the number of combinations in which there are 0  $w_0$  plays, then 1  $w_0$  plays, etc.
- 15.6: If  $i > 0$  and  $j > 0$ , you can get to  $(i, j)$  from  $(i - 1, j)$  or  $(j - 1, i)$ .
- 15.7: Reduce the problem from  $n$  symbols to one on  $n - 1$  symbols.
- 16.1: Model the maze as a graph.
- 16.2: Treat strings as vertices in an undirected graph, with an edge between  $u$  and  $v$  iff the corresponding strings differ in one character.
- 16.3: Form a DAG in which paths correspond to valid placements.
- 16.4: What property does a minimal set of infeasible tasks have?
- 16.5: Change the edge cost and cast it as an instance of the standard shortest path problem.
- 17.1: Consider the Boolean-valued function  $P(i, w)$ , which is true iff it is possible for the first  $i$  states to assign  $w$  votes to  $R$ .
- 17.2: The “obvious” recurrence is not the right one.
- 17.3: Solve the  $n$  jugs case.
- 17.4: Apply the constraints to speed up a brute-force algorithm.
- 18.1: Look for races, and lock as little as possible to avoid reducing throughput.
- 18.2: There are two aspects—data structure design and concurrency.
- 18.3: Track the number of readers.
- 19.1: How would you define the distance between two images?
- 19.2: This must be performed on an ensemble of machines. The right data structures will simplify the computation.
- 20.1: Use a routine that yields  $k$ -sized subsets to create a routine for  $k + 1$ -sized subsets.
- 20.2: How would you mimic a three-sided coin with a two-sided coin?
- 20.3: 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?
- 20.4: Simulate Solution 20.1 on Page 190, using an appropriate data structure to reduce space.
- 21.1: Solve the same problem with 2, 5, 10, and 20 doors.
- 21.2: Use transitivity to eliminate as many cyclists with each race as possible.
- 21.3: Think about starting with more than enough gas to complete the circuit without gassing up. Track the amount of gas as you perform the circuit, gassing up at each city.
- 21.4: Consider the products of the first  $i - 1$  and the last  $n - i$  elements. Alternately, count the number of negative entries and zero entries.
- 21.5: Write a recurrence relation.
- 21.6: Model celebrities and bidders as vertices.

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. See [Solution 13.1](#) on [Page 152](#) for an example.
- 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 12.3](#) on [Page 150](#) 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. 46:** *How would you go about computing the parity of a very large number of 64-bit nonnegative integers?*

**Solution 5.1:** The fastest algorithm for manipulating bits can vary based on the underlying hardware.

The time taken to directly compute the parity of a single number is proportional to the number of bits:

```
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 }
```

A neat trick that erases the lowest set bit of a number in a single operation 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); // drops the lowest set bit of x.
6     }
7     return result;
8 }
```

However, when you have to perform a large number of parity operations, and more generally, any kind of bit fiddling operation, the best way to proceed is to precompute the answer and store it in an array. Depending upon how much memory is at your disposal, and how much fits efficiently in cache, you can vary the size of the lookup table. Below is an example implementation where you build a lookup table “`precomputed_parity`” that stores the parity of any 16-bit number *i* as `precomputed_parity[i]`. This array can either be constructed during static initialization or dynamically—a flag bit can be used to indicate if the entry at a location is uninitialized. Once you have this array, you can implement the parity function as follows:

```
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  $\langle b_{63}, b_{62}, \dots, b_3, b_2, b_1, b_0 \rangle$  equals the parity of  $\langle b_{63} \oplus b_{31}, b_{62} \oplus b_{30}, \dots, b_{32} \oplus b_0 \rangle$ ; 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. 46:** Implement a method that takes as input a set  $S$  of distinct elements, and prints the power set of  $S$ . Print the subsets one per line, with elements separated by commas.

**Solution 5.2:** The key to solving this problem is realizing that for a given ordering of the elements of  $S$ , there exists a one-to-one correspondence between the  $2^{|S|}$  bit arrays of length  $|S|$  and the set of all subsets of  $S$ —the 1s in the  $|S|$ -length bit array  $v$  indicate the elements of  $S$  in the subset corresponding to  $v$ .

For example, if  $S = \{g, l, e\}$  and the elements are ordered  $g < l < e$ , the bit array  $\langle 0, 1, 1 \rangle$  denotes the subset  $\{l, e\}$ .

If  $|S|$  is less than or equal to the number of bits used to represent an integer on the architecture (or language) we are working on, we can enumerate bit arrays by enumerating integers in  $[0, 2^{|S|} - 1]$  and examining the indices of bits set in these inte-

gers. These indices are determined by first isolating the lowest set bit by computing  $y = x \& \sim(x - 1)$ , which is described on Page 4 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 }

```

In practice, it would likely be faster to iterate through all the bits in  $x$ , one at a time.

Alternately, we can use recursion. We make one call with the  $i$ -th element and one call without the  $i$ -th element. The time complexity is  $O(|S|2^{|S|})$ . The space complexity is  $O(|S|)$  which comes from the maximum stack depth as well as the maximum size of a subset.

```

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

```

**Variant 5.2.1:** Print all subsets of size  $k$  of  $\{1, 2, 3, \dots, n\}$ .

**Problem 5.3, pg. 47:** Implement string/integer inter-conversion functions. Use the following function signatures: `String intToString(int x)` and `int stringToInt(String s)`.

**Solution 5.3:** For a positive integer  $x$ , we iteratively divide  $x$  by 10, and record the remainder till we get to 0. This yields the result from the least significant digit, and needs to be reversed. If  $x$  is negative, we record that, and negate  $x$ , adding a '-' afterward. If  $x$  is 0, our code breaks out of the iteration without writing any digits, in which case we need to explicitly set a 0. In C++ code:

```

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 5.4, pg. 47:** Design an efficient algorithm for computing the GCD of two numbers without using multiplication, division or the modulus operators.

**Solution 5.4:** 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 lengths of the arguments.

```

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 5.5, pg. 48:** 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 5.5:** 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  $O(n)$ , assuming constant time arithmetic. 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 6.1, pg. 49:** Write a function that takes an array  $A$  and an index  $i$  into  $A$ , and rearranges the elements such that all elements less than  $A[i]$  appear first, followed by elements equal to  $A[i]$ , followed by elements greater than  $A[i]$ . Your algorithm should have  $O(1)$  space complexity and  $O(|A|)$  time complexity.

**Solution 6.1:** This problem is conceptually straightforward: maintain four groups, *bottom* (elements less than pivot), *middle* (elements equal to pivot), *unclassified*, and *top* (elements greater than pivot). These groups are stored in contiguous order in  $A$ . To make this partitioning run in  $O(1)$  space, we use *smaller*, *equal*, and *larger* pointers to track these groups in the following way:

- *bottom*: stored in subarray  $A[0 : \text{smaller} - 1]$ .
- *middle*: stored in subarray  $A[\text{smaller} : \text{equal} - 1]$ .
- *unclassified*: stored in subarray  $A[\text{equal} : \text{larger}]$ .
- *top*: stored in subarray  $A[\text{larger} + 1 : |A| - 1]$ .

We explore elements of *unclassified* in order, and classify the element into one of *bottom*, *middle*, and *top* groups according to the relative order between the incoming unclassified element and pivot. Each iteration decreases the size of *unclassified* group by 1, and the time spent within each iteration is constant, implying the time complexity is  $\Theta(|A|)$ .

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

```

1 void dutch_flag_partition(vector<int>* A, int pivot_index) {
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 }

```

**$\epsilon$ -Variant 6.1.1:** Assuming that keys take one of three values, reorder the array so that all objects of the same key appear in the same subarray. The order of the subarrays is not important. For example, both Figures 6.1(b) and 6.1(c) on Page 50 are valid answers for Figure 6.1(a) on Page 50. Use  $O(1)$  additional space and  $O(|A|)$  time.

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

**$\epsilon$ -Variant 6.1.3:** Given an array  $A$  of objects with Boolean-valued keys, reorder the array so that all objects that have the same key appear in the same subarray. Use  $O(1)$  additional space and  $O(|A|)$  time.

**Problem 6.2, pg. 50:** 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  $z_0, \dots, z_{n-1}$ . Assume that the battery capacity is such that with the fully charged battery, the robot can climb  $B$  meters. The robot will run out of energy iff there exist integers  $i$  and  $j$  such that  $i < j$  and  $z_j - z_i > B$ , i.e., to go from Point  $i$  to Point  $j$ , the robot has to climb more than  $B$  meters. Therefore, we would like to pick  $B$  such that for any  $i < j$ , we have  $B \geq z_j - z_i$ .

We developed several algorithms for this problem in the introduction. Specifically, on Page 2 we showed how to compute the minimum  $B$  in  $O(n)$  time by keeping the running min as we do a sweep. In code:

```

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 }

```

**Problem 6.3, pg. 51:** 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  $i_0 < j_0 < i_1 < j_1 < \dots < i_{k-1} < j_{k-1}$ . 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:

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

The key to achieving an  $O(kn)$  time bound is the observation that computing  $B$  and  $S$  requires computing  $\max_{i' < i} B_{i'}^{j-1}$  and  $\max_{i' < i} S_{i'}^{j-1}$ . These two quantities can be computed in constant time for each  $i$  and  $j$  with a conditional update. In code:

```

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 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.

```

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 }

```

**Problem 6.4, pg.51:** Implement a function for reversing the words in a string. Your function should use  $O(1)$  space.

**Solution 6.4:** The code for computing the position for each character in a single pass is fairly complex. However, a two stage iteration is easy. In the first step, reverse the entire string and in the second step, reverse each word. For example, “ram is costly” transforms to “yltsoc si mar”, which transforms to “costly is ram”. Here is code in C++:

```

1 void reverse_words(string* input) {

```

```

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

```

**Problem 6.5, pg. 51:** Design an algorithm that takes a string  $s$  and a string  $r$ , assumed to be a well-formed ESRE, and checks if  $r$  matches  $s$ .

**Solution 6.5:** The key to solving this problem is using recursion effectively.

If  $r$  starts with  $^$ , then the remainder of  $r$ , i.e.,  $r^1$ , must strictly match a prefix of  $s$ . If  $r$  ends with a  $\$$ , some suffix of  $s$  must be strictly matched by  $r$  without the trailing  $\$$ . Otherwise,  $r$  must strictly match some substring of  $s$ .

Call the function that checks whether  $r$  strictly matches a prefix of string  $s$  `is_match`. This function has to check several cases:

- (1.) Length-0 ESREs which match everything.
- (2.) An ESRE starting with  $^$  or ending with  $\$$ .
- (3.) An ESRE starting with an alphanumeric character or dot.
- (4.) An ESRE starting with a  $*$  match, e.g.,  $a^*wXY$  or  $.*Wa$ .

Case (1.) is a base case. Case (2.) involves a check possibly followed by a recursive call to `is_match_here`. Case (3.) requires a single call to `is_match_here`. Case (4.) is handled by a walk down the string  $s$ , checking that the prefix of  $s$  thus far matches the alphanumeric character or dot until some suffix of  $s$  is matched by the remainder of the ESRE, i.e.,  $r^2$ .

```

1 bool is_match(const string &r, const string &s) {
2     // Case (2.) : starts with '^'.
3     if (r.front() == '^') {
4         return is_match_here(r.substr(1), s);
5     }
6
7     for (int i = 0; i <= s.size(); ++i) {
8         if (is_match_here(r, s.substr(i))) {
9             return true;
10        }
11    }
12    return false;
13 }
14
15 bool is_match_here(const string &r, const string &s) {
16     // Case (1.)
17     if (r.empty()) {
18         return true;
19    }

```

```

20
21 // Case (2) : ends with '$'.
22 if (r == "$") {
23     return s.empty();
24 }
25
26 // Case (4.)
27 if (r.size() >= 2 && r[1] == '*') {
28     for (string::size_type i = 0;
29         i < s.size() && (r.front() == '.' || r.front() == s[i]);
30         ++i) {
31         if (is_match_here(r.substr(2), s.substr(i + 1))) {
32             return true;
33         }
34     }
35     return is_match_here(r.substr(2), s);
36 }
37
38 // Case (3.)
39 return !s.empty() && (r.front() == '.' || r.front() == s.front()) &&
40     is_match_here(r.substr(1), s.substr(1));
41 }

```

**ε-Variant 6.5.1:** Solve the same problem for regular expressions without the ^ and \$ operators.

**Problem 7.1, pg. 54:** Write a function that takes *L* and *F*, and returns the merge of *L* and *F*. Your code should use  $O(1)$  additional storage—it should reuse the nodes from the lists provided as input. Your function should use  $O(1)$  additional storage, as illustrated in Figure 7.3 on Page 54. The only field you can change in a node is *next*.

**Solution 7.1:** We traverse the lists, using one pointer per list, each initialized to the list head. We compare the contents of the pointer—the pointer with the lesser contents is to be added to the end of the result and advanced. If either pointer is null, we add the sublist pointed to by the other to the end of the result. The add can be performed by a single pointer update—it does not entail traversing the sublist. The worst case time complexity corresponds to the case when the lists are of comparable length. In the best case, one list is much shorter than the other and all its entries appear at the beginning of the merged list.

```

1 template <typename T>
2 shared_ptr<node_t<T>> merge_sorted_linked_lists(shared_ptr<node_t<T>> F,
3                                                 shared_ptr<node_t<T>> L) {
4     shared_ptr<node_t<T>> sorted_head = nullptr, tail = nullptr;
5
6     while (F && L) {
7         append_node_and_advance(&sorted_head, &tail, F->data < L->data ? &F : &L);
8     }
9
10    // Append the remaining nodes of F.
11    if (F) {

```

```

12     append_node(&sorted_head, &tail, &F);
13 }
14 // Append the remaining nodes of L.
15 if (L) {
16     append_node(&sorted_head, &tail, &L);
17 }
18 return sorted_head;
19 }
20
21 template <typename T>
22 void append_node_and_advance(shared_ptr<node_t<T>>* head,
23                             shared_ptr<node_t<T>>* tail,
24                             shared_ptr<node_t<T>>* n) {
25     append_node(head, tail, n);
26     *n = (*n)->next; // advance n.
27 }
28
29 template <typename T>
30 void append_node(shared_ptr<node_t<T>>* head,
31                 shared_ptr<node_t<T>>* tail,
32                 shared_ptr<node_t<T>>* n) {
33     *head ? (*tail)->next = *n : *head = *n;
34     *tail = *n; // reset tail to the last node.
35 }

```

**ε-Variant 7.1.1:** Solve the same problem when the lists are doubly linked.

**Problem 7.2, pg. 54:** Given a reference to the head of a singly linked list *L*, how would you determine whether *L* ends in a null or reaches a cycle of nodes? Write a function that returns null if there does not exist a cycle, and the reference to the start of the cycle if a cycle is present. (You do not know the length of the list in advance.)

**Solution 7.2:** This problem has several solutions. If space is not an issue, the simplest approach is to explore nodes via the next field starting from the head and storing visited nodes in a hash table—a cycle exists iff we visit a node already in the hash table. If no cycle exists, the search ends at the tail (often represented by having the next field set to null). This solution requires  $\Theta(n)$  space, where  $n$  is the number of nodes in the list.

In some languages, e.g., C, the next field is a pointer. Typically, for performance reasons related to the memory subsystem on a processor, memory is allocated on word boundaries, and (at least) two of the least significant bits in the next pointer are 0. Bit fiddling can be used to set the least significant bit on the next pointer to mark whether a node has been visited. This approach has the disadvantage of changing the data structure—these updates can be undone later.

Another approach is to reverse the linked list, in the manner of Solution 7.4 on Page 126. If the head is encountered during the reversal, it means there is a cycle; otherwise we will get to the tail. Although this approach requires no additional storage, and runs in  $O(n)$  time, it does modify the list.

A naïve approach that does not use additional storage and does not modify the list is to walk the list in two loops—the outer loop visits the nodes one-by-one, and the inner loop starts from the head, and visits  $m$  nodes, where  $m$  is the number of nodes visited in the outer loop. If the node being visited by the outer loop is visited twice, a loop has been detected. (If the outer loop encounters the end of the list, no cycle exists.) This approach has  $O(n^2)$  time complexity.

This idea can be made to work in linear time—use a slow pointer, *slow*, and a fast pointer, *fast*, to visit the list. In each iteration, advance *slow* by one and *fast* by two. The list has a cycle iff the two pointers meet.

This is proved as follows.

**Proof:**

Number the nodes in the cycle by assigning first node encountered the index 0. Let  $C$  be the total number of nodes in the cycle. If the fast pointer reaches the first node at iteration  $F$ , at iteration  $i \geq F$ , it will be at node  $2(i - F) \bmod C$ . If the slow pointer reaches the first node at iteration  $S$ , at iteration  $i \geq S$ , it will be at node  $(i - S) \bmod C$ . The difference between the pointer locations after the slow pointer reaches the first node in the cycle is  $2(i - F) - (i - S) \bmod C = i - (2F - S) \bmod C$ . As  $i$  increases by one in each iteration, the equation  $(i - (2F - S)) \bmod C = 0$  has a solution.

Now, assuming that we have detected a cycle using the above method, we find the start of the cycle, by first calculating the cycle length. We do this by freezing the fast pointer, and counting the number of times we have to advance the slow pointer to come back to the fast pointer. Consequently, we set both *slow* and *fast* pointers to the head. Then we advance *fast* by the length of the cycle, then move both *slow* and *fast* one at a time. The start of the cycle is located at the node where these two pointers meet again.

The code to do this traversal is quite simple in C++:

```

1  template <typename T>
2  shared_ptr<node_t<T>> has_cycle(const shared_ptr<node_t<T>> & head) {
3      shared_ptr<node_t<T>> fast = head, slow = head;
4
5      while (slow && slow->next && fast && fast->next && fast->next->next) {
6          slow = slow->next, fast = fast->next->next;
7          if (slow == fast) { // there is a cycle.
8              // Calculates the cycle length.
9              int cycle_len = 0;
10             do {
11                 ++cycle_len;
12                 fast = fast->next;
13             } while (slow != fast);
14
15             // Tries to find the start of the cycle.
16             slow = head, fast = head;
17             // Fast pointer advances cycle_len first.
18             while (cycle_len--) {
19                 fast = fast->next;
20             }

```



```

21     // Both pointers advance at the same time.
22     while (slow != fast) {
23         slow = slow->next, fast = fast->next;
24     }
25     return slow; // the start of cycle.
26 }
27 }
28 return nullptr; // no cycle.
29 }

```

**ε-Variant 7.2.1:** The following program purports to compute the beginning of the cycle without determining the length of the cycle; it has the benefit of being more succinct than the code listed above. Is the program correct?

```

1  template <typename T>
2  shared_ptr<node_t<T>> has_cycle(const shared_ptr<node_t<T>>& head) {
3      shared_ptr<node_t<T>> fast = head, slow = head;
4
5      while (slow && slow->next && fast && fast->next && fast->next->next) {
6          slow = slow->next, fast = fast->next->next;
7          if (slow == fast) { // there is a cycle.
8              // Tries to find the start of the cycle.
9              slow = head;
10             // Both pointers advance at the same time.
11             while (slow != fast) {
12                 slow = slow->next, fast = fast->next;
13             }
14             return slow; // slow is the start of cycle.
15         }
16     }
17     return nullptr; // means no cycle.
18 }

```

**Problem 7.3, pg. 54:** Let  $h_1$  and  $h_2$  be the heads of lists  $L_1$  and  $L_2$ , respectively. Assume that  $L_1$  and  $L_2$  are well-formed, that is each consists of a finite sequence of nodes. (In particular, neither list has a cycle.) How would you determine if there exists a node  $r$  reachable from both  $h_1$  and  $h_2$  by following the next fields? If such a node exists, find the node that appears earliest when traversing the lists. You are constrained to use no more than constant additional storage.

**Solution 7.3:** The lists overlap iff both have the same tail node: since each node has a single next field, once the lists converge at a node, they cannot diverge at a later node. Let  $|L|$  denote the number of nodes in list  $L$ . Checking overlap amounts to finding the tail nodes for each, which is easily performed in  $O(|L_1| + |L_2|)$  time and  $O(1)$  space. To find the first node, we proceed as above, and in addition we compute  $|L_1|$  and  $|L_2|$ . The first node is determined by first advancing through the longer list by  $||L_1| - |L_2||$  nodes, and then advancing through both lists in lock-step, stopping at the first common node.

```

1 template <typename T>
2 shared_ptr<node_t<T>> overlapping_no_cycle_lists(shared_ptr<node_t<T>> L1,
3                                                shared_ptr<node_t<T>> L2) {
4     // Count the lengths of L1 and L2.
5     int L1_len = count_len(L1), L2_len = count_len(L2);
6
7     // Advance the longer list.
8     advance_list_by_k(L1_len > L2_len ? &L1 : &L2, abs(L1_len - L2_len));
9
10    while (L1 && L2 && L1 != L2) {
11        L1 = L1->next, L2 = L2->next;
12    }
13    return L1; // nullptr means no overlap between L1 and L2.
14 }
15
16 // Counts the list length till end.
17 template <typename T>
18 int count_len(shared_ptr<node_t<T>> L) {
19     int len = 0;
20     while (L) {
21         ++len, L = L->next;
22     }
23     return len;
24 }
25
26 template <typename T>
27 void advance_list_by_k(shared_ptr<node_t<T>>* L, int k) {
28     while (k--) {
29         *L = (*L)->next;
30     }
31 }

```

Figure 22.1 shows an example of lists which overlap and have cycles. For this example, both *A* and *B* are acceptable answers.

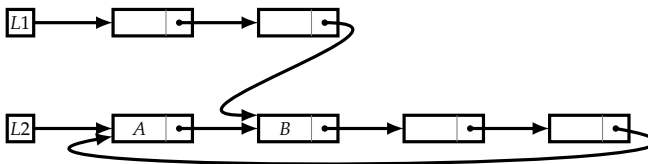


Figure 22.1: Overlapping lists.

**Problem 7.4, pg. 55:** Give a linear time non-recursive function that reverses a singly linked list. The function should use no more than constant storage beyond that needed for the list itself.

**Solution 7.4:** The natural way of implementing the reversal is through recursion. However, this approach implicitly uses  $\Theta(n)$  space on the stack. The function is not

tail recursive, which precludes compilers from automatically converting the function to an iterative one.

Reversal can be performed iteratively—walk the list with two pointers, and update the trailing pointer's next field. It uses  $O(1)$  additional storage, and has  $\Theta(n)$  time complexity.

Recursive implementation, uses  $\Theta(n)$  storage on the function call stack:

```

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

```

Iterative implementation:

```

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

```

**Problem 7.5, pg. 55:** Implement a function which takes as input a pointer to the head of a postings list  $L$ , and returns a copy of the postings list. Your function should take  $O(n)$  time, where  $n$  is the length of the postings list and should use  $O(1)$  storage beyond that required for the  $n$  nodes in the copy. You can modify the original list, but must restore it to its initial state before returning.

**Solution 7.5:** 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 the next page on Page

on Pages

Code implementing the copy is given below:

```

1 template <typename T>
2 shared_ptr<node_t<T>> copy_postings_list(const shared_ptr<node_t<T>>& L) {
3     // Return empty list if L is nullptr.
4     if (!L) {
5         return nullptr;
6     }
7
8     // 1st stage: Copy the nodes from L.
9     shared_ptr<node_t<T>> p = L;
10    while (p) {
11        auto temp = make_shared<node_t<T>>(node_t<T>{p->data, p->next, nullptr});
12        p->next = temp;
13        p = temp->next;
14    }
15
16    // 2nd stage: Update the jump field.
17    p = L;
18    while (p) {
19        if (p->jump) {
20            p->next->jump = p->jump->next;
21        }
22        p = p->next->next;
23    }
24
25    // 3rd stage: Restore the next field.
26    p = L;
27    shared_ptr<node_t<T>> copied = p->next;
28    while (p->next) {
29        shared_ptr<node_t<T>> temp = p->next;
30        p->next = temp->next;
31        p = temp;
32    }
33    return copied;
34 }

```

**Problem 8.1, pg. 56:** Design a stack that supports a **max** operation, which returns the maximum value stored in the stack, and throws an exception if the stack is empty. Assume elements are comparable. All operations must be  $O(1)$  time. You can use  $O(n)$  additional space, beyond what is required for the elements themselves.

**Solution 8.1:** A conceptually straightforward approach to tracking the maximum is store pairs in a stack. The first component is the key being pushed; the second is the largest value in the stack after the push is completed. When we push a value, the

maximum value stored at or below any of the entries below the entry just pushed does not change. The pushed entry's maximum value is simply the larger of the value just pushed and the maximum prior to the push, which can be determined by inspecting the maximum field of the element below. Since popping does not change the values below, there is nothing special to be done for pop. Of course appropriate checks have to be made to ensure the stack is not empty.

This approach has  $O(1)$  time complexity for the specified methods. The additional space complexity is  $\Theta(n)$ , regardless of the stored keys.

```
1 template <typename T>
2 class Stack {
3 public:
4     bool empty() const { return s_.empty(); }
5
6     const T& max() const {
7         if (!empty()) {
8             return s_.top().second;
9         }
10        throw length_error("empty stack");
11    }
12
13    T pop() {
14        if (empty()) {
15            throw length_error("empty stack");
16        }
17        T ret = s_.top().first;
18        s_.pop();
19        return ret;
20    }
21
22    void push(const T& x) {
23        s_.emplace(x, std::max(x, empty() ? x : s_.top().second));
24    }
25
26 private:
27     stack<pair<T, T>> s_;
28 };
```

Heuristically, the additional space required can be reduced by maintaining two stacks, the primary stack, which holds the keys being pushed, and an auxiliary stack, whose operation we now describe.

The top of the auxiliary stack holds a pair. The first component of the pair is the maximum key in the primary stack. The second component is the number of times that key appears in the primary stack.

Let  $m$  be the maximum key currently in the primary stack. There are three cases to consider when a key  $k$  is pushed.

1.  $k$  is smaller than  $m$ . The auxiliary stack is not updated.
2.  $k$  is equal to  $m$ . We increment the second component of the pair stored at the top of the auxiliary stack.
3.  $k$  is greater than  $m$ . The pair  $(k, 1)$  is pushed onto the auxiliary stack.

There are two cases to consider when the primary stack is popped. Let  $k$  be the popped key.

1.  $k$  is less than  $m$ . The auxiliary stack is not updated.
2.  $k$  is equal to  $m$ . We decrement the second component of the top of the auxiliary stack. If its value becomes 0, we pop the auxiliary stack.

These operations are illustrated in Figure 22.2 on the facing page.

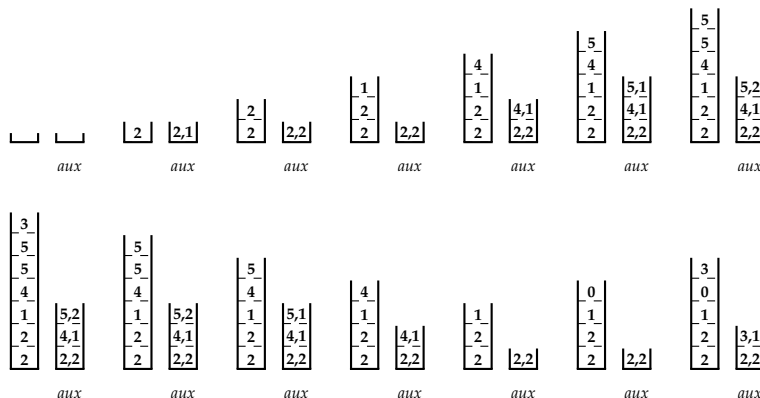
```

1  template <typename T>
2  class Stack {
3  public:
4      bool empty() const { return s_.empty(); }
5
6      const T& max() const {
7          if (!empty()) {
8              return aux_.top().first;
9          }
10         throw length_error("empty stack");
11     }
12
13     T pop() {
14         if (empty()) {
15             throw length_error("empty stack");
16         }
17         T ret = s_.top();
18         s_.pop();
19         if (ret == aux_.top().first) {
20             --aux_.top().second;
21             if (aux_.top().second == 0) {
22                 aux_.pop();
23             }
24         }
25         return ret;
26     }
27
28     void push(const T& x) {
29         s_.emplace(x);
30         if (!aux_.empty()) {
31             if (x == aux_.top().first) {
32                 ++aux_.top().second;
33             } else if (x > aux_.top().first) {
34                 aux_.emplace(x, 1);
35             }
36         } else {
37             aux_.emplace(x, 1);
38         }
39     }
40
41 private:
42     stack<T> s_;
43     stack<pair<T, int>> aux_;
44 };

```

The worst-case additional space complexity is  $\Theta(n)$ , which occurs when each key pushed is greater than all keys in the primary stack. However, when the number

of distinct keys is small, or the maximum changes infrequently, the additional space complexity is less,  $O(1)$  in the best case. The time complexity for each specified method is still  $O(1)$ .



**Figure 22.2:** 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 8.2, pg. 57:** Given the root node *r* of a binary tree, print all the keys in level order at *r* and its descendants. Specifically, the nodes should be printed in order of their level, with all keys at a given level in a single line, and the next line corresponding to keys at the next level. 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 9.1 on Page 59.

**Solution 8.2:** We maintain a queue of nodes to process. Specifically the queue contains nodes at level *l* followed by nodes at level *l* + 1. After all nodes from level *l* are processed, the head of the queue is a node at level *l* + 1; processing this node introduces nodes from level *l* + 2 to the end of the queue. We use a count variable that records the number of nodes at the level of the head of the queue that remain to be processed. When all nodes at level *l* are processed, the queue consists of exactly the set of nodes at level *l* + 1, and count is updated to the size of the queue.

```
1 template <typename T>
2 void print_binary_tree_level_order(const unique_ptr<BinaryTree<T>>& n) {
```

```

3 // Prevent empty tree.
4 if (!n) {
5     return;
6 }
7
8 queue<BinaryTree<T>*> q;
9 q.emplace(n.get());
10 size_t count = q.size();
11 while (!q.empty()) {
12     cout << q.front()->data << ' ';
13     if (q.front()->left) {
14         q.emplace(q.front()->left.get());
15     }
16     if (q.front()->right) {
17         q.emplace(q.front()->right.get());
18     }
19     q.pop();
20     if (--count == 0) {
21         cout << endl;
22         count = q.size();
23     }
24 }
25 }

```

**Problem 8.3, pg. 58:** *Implement a queue API using an array for storing elements. Your API should include a constructor function, which takes as argument the capacity of the queue, enqueue and dequeue functions, a size function, which returns the number of elements stored, and implement dynamic resizing.*

**Solution 8.3:** 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 an enqueue is attempted we resize. When count = 0 and a dequeue is attempted we throw an exception.

```

1 template <typename T>
2 class Queue {
3 public:
4     explicit Queue(size_t cap) : data_(cap) {}
5
6     void enqueue(const T& x) {
7         // Dynamically resize due to data_.size() limit.
8         if (count_ == data_.size()) {
9             // Rearrange elements.
10            rotate(data_.begin(), data_.begin() + head_, data_.end());
11            head_ = 0, tail_ = count_; // reset head and tail.
12            data_.resize(data_.size() << 1);
13        }
14        // Perform enqueue.
15        data_[tail_] = x;

```



```

16     tail_ = (tail_ + 1) % data_.size(), ++count_;
17 }
18
19 T dequeue() {
20     if (count_) {
21         --count_;
22         T ret = data_[head_];
23         head_ = (head_ + 1) % data_.size();
24         return ret;
25     }
26     throw length_error("empty queue");
27 }
28
29 size_t size() const { return count_; }
30
31 private:
32     size_t head_ = 0, tail_ = 0, count_ = 0;
33     vector<T> data_;
34 };

```

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 9.1, pg. 61:** Write a function that takes as input the root of a binary tree and returns `true` or `false` depending on whether the tree is symmetric.

**Solution 9.1:** We present a recursive algorithm that follows directly from the definition of symmetry.

```

1  template <typename T>
2  bool is_symmetric(const unique_ptr<BinaryTree<T>>& n) {
3      return !n || is_symmetric_helper<T>(n->left, n->right);
4  }
5
6  template <typename T>
7  bool is_symmetric_helper(const unique_ptr<BinaryTree<T>>& l,
8                          const unique_ptr<BinaryTree<T>>& r) {
9      if (!l && !r) {
10         return true;
11     } else if (l && r) {
12         return l->data == r->data && is_symmetric_helper<T>(l->left, r->right) &&
13             is_symmetric_helper<T>(l->right, r->left);
14     } else { // (l && !r) || (!l && r)
15         return false;
16     }
17 }

```

**Problem 9.2, pg. 61:** Let  $T$  be the root of a binary tree in which nodes have an explicit parent field. Design an iterative algorithm that enumerates the nodes inorder and uses  $O(1)$  additional space. Your algorithm cannot modify the tree.

**Solution 9.2:** The standard idiom for an inorder walk is visit-left, visit-root, visit-right. Accessing the left child is straightforward. Returning from a left child  $l$  to its parent entails examining  $l$ 's parent field; returning from a right child  $r$  to its parent is similar.

To make this scheme work, we need to know when we take a parent pointer to node  $r$  if the child we completed visiting was  $r$ 's left child (in which case we need to visit  $r$  and then  $r$ 's right child) or a right child (in which case we have completed visiting  $r$ ). We achieve this by storing the child in a `prev` variable before we move to the parent,  $r$ . We then compare `prev` with  $r$ 's left child and the right child.

```

1  template <typename T>
2  void inorder_traversal(const unique_ptr<BinaryTree<T>>& r) {
3      // Empty tree.
4      if (!r) {
5          return;
6      }
7
8      BinaryTree<T>* prev = nullptr, *curr = r.get(), *next;
9      while (curr) {
10         if (!prev || prev->left.get() == curr || prev->right.get() == curr) {
11             if (curr->left) {
12                 next = curr->left.get();
13             } else {
14                 cout << curr->data << endl;
15                 next = (curr->right ? curr->right.get() : curr->parent);
16             }
17         } else if (curr->left.get() == prev) {
18             cout << curr->data << endl;
19             next = (curr->right ? curr->right.get() : curr->parent);
20         } else { // curr->right.get() == prev.
21             next = curr->parent;
22         }
23
24         prev = curr;
25         curr = next;
26     }
27 }
```

**ε-Variant 9.2.1:** How would you perform preorder and postorder walks iteratively using  $O(1)$  additional space? Your algorithm cannot modify the tree. Nodes have an explicit parent field.

**Problem 9.3, pg. 62:** 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 9.3:** 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 walk and has no successor.

```

1  template <typename T>
2  BinaryTree<T>* find_successor(const unique_ptr<BinaryTree<T>>& 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 }

```

**Problem 9.4, pg. 62:** Design an efficient algorithm for computing the LCA of nodes  $a$  and  $b$  in a binary tree in which nodes do not have a parent pointer.

**Solution 9.4:** Let  $a$  and  $b$  be the nodes whose LCA we wish to compute. Observe that if the root is one of  $a$  or  $b$ , then it is the LCA. Otherwise, let  $L$  and  $R$  be the trees rooted at the left child and the right child of the root. If both nodes lie in  $L$  (or  $R$ ), their LCA is in  $L$  (or  $R$ ). Otherwise, their LCA is the root itself. This is the basis for the algorithm presented below. Its time complexity is  $O(n)$ , where  $n$  is the number of nodes.

```

1  template <typename T>
2  BinaryTree<T>* LCA(const unique_ptr<BinaryTree<T>>& n,
3                    const unique_ptr<BinaryTree<T>>& a,
4                    const unique_ptr<BinaryTree<T>>& b) {
5      if (!n) { // empty subtree.
6          return nullptr;
7      } else if (n == a || n == b) {
8          return n.get();
9      }
10
11     auto* l_res = LCA(n->left, a, b), *r_res = LCA(n->right, a, b);
12     if (l_res && r_res) {
13         return n.get(); // found a and b in different subtrees.
14     } else {
15         return l_res ? l_res : r_res;
16     }
17 }

```

**Problem 10.1, pg. 63:** 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 10.1:** 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,
3                     const pair<int, int>& rhs) const {
4         return lhs.first > rhs.first;
5     }
6 };
7
8 vector<int> merge_arrays(const vector<vector<int>>& S) {
9     priority_queue<pair<int, int>, vector<pair<int, int>>, Compare> min_heap;
10    vector<int> S_idx(S.size(), 0);
11
12    // Every array in S puts its smallest element in heap.
13    for (int i = 0; i < S.size(); ++i) {
14        if (S[i].size() > 0) {
15            min_heap.emplace(S[i][0], i);
16            S_idx[i] = 1;
17        }
18    }
19
20    vector<int> ret;
21    while (!min_heap.empty()) {
22        pair<int, int> p = min_heap.top();
23        ret.emplace_back(p.first);
24        // Add the smallest element into heap if possible.
25        if (S_idx[p.second] < S[p.second].size()) {
26            min_heap.emplace(S[p.second][S_idx[p.second++]], p.second);
27        }
28        min_heap.pop();
29    }
30    return ret;
31 }

```

**Problem 10.2, pg. 64:** Design an efficient algorithm for sorting a  $k$ -increasing-decreasing array. You are given another array of the same size that the result should be written to, and

you can use  $O(k)$  additional storage.

**Solution 10.2:** The first thing to note is that any array can be decomposed into a sequence of increasing and decreasing subarrays. If  $k$  is comparable to  $n$ , then the problem is equivalent to the general sorting problem.

If  $k$  is substantially smaller than  $n$ , we could first reverse the order of the decreasing subarrays. Now we can use the techniques in Solution 10.1 on the facing page to sort the array in time  $O(n \log k)$  time with  $O(k)$  space.

```

1 vector<int> sort_k_increasing_decreasing_array(const vector<int>& A) {
2     // Decompose A into a set of sorted arrays.
3     vector<vector<int>> S;
4     bool is_increasing = true; // the trend we are looking for.
5     int start_idx = 0;
6     for (int i = 1; i < A.size(); ++i) {
7         if ((A[i - 1] < A[i] && !is_increasing) ||
8             (A[i - 1] >= A[i] && is_increasing)) {
9             if (is_increasing) {
10                S.emplace_back(A.cbegin() + start_idx, A.cbegin() + i);
11            } else {
12                S.emplace_back(A.crbegin() + A.size() - i,
13                              A.crbegin() + A.size() - start_idx);
14            }
15            start_idx = i;
16            is_increasing = !is_increasing; // inverse the trend.
17        }
18    }
19    if (start_idx < A.size()) {
20        if (is_increasing) {
21            S.emplace_back(A.cbegin() + start_idx, A.cend());
22        } else {
23            S.emplace_back(A.crbegin(), A.crbegin() + A.size() - start_idx);
24        }
25    }
26
27    return merge_arrays(S);
28 }

```

**Problem 10.3, pg. 64:** How would you compute the  $k$  stars which are closest to the Earth? You have only a few megabytes of RAM.

**Solution 10.3:** 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  $O(k)$ .

```

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(istream& sin, int k) {
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 }

```

**Problem 10.4, pg. 64:** *The input consists of a very long sequence of numbers. Each number is at most  $k$  positions away from its correctly sorted position. Design an algorithm that outputs the numbers in the correct order and uses  $O(k)$  storage, independent of the number of elements processed.*

**Solution 10.4:** The easiest way of looking at this problem is that we need to store the numbers in memory till all the numbers smaller than this number have arrived. Once those numbers have arrived and have been written to the output file, we can go ahead and write this number. Since we do not know precisely what order the numbers appear in, it is not possible to say when all the numbers smaller than a given number have arrived and have been written to the output. However since we are told that no number is off by more than  $k$  positions from its correctly sorted position, if more than  $k$  numbers greater than a given number have arrived and all the numbers smaller than the given number that arrived have been written, we can be sure that there are no more other smaller numbers that are going to arrive. Hence it is safe to write the given numbers.

This essentially gives us the strategy to always keep  $k + 1$  numbers in a min-heap. As soon as we read a new number, we extract the min from the heap and write the output and then insert the new number.

```
1 void approximate_sort(istream* sin, int k) {
2     priority_queue<int, vector<int>, greater<int>> min_heap;
3     // Firstly push k elements into min_heap.
4     int x;
5     for (int i = 0; i < k && *sin >> x; ++i) {
6         min_heap.push(x);
7     }
8
9     // Extract the minimum one for every incoming element.
10    while (*sin >> x) {
11        min_heap.push(x);
12        cout << min_heap.top() << endl;
13        min_heap.pop();
14    }
15
16    // Extract the remaining elements in min_heap.
17    while (!min_heap.empty()) {
18        cout << min_heap.top() << endl;
19        min_heap.pop();
20    }
21 }
```

**Problem 10.5, pg. 64:** *Design an algorithm for efficiently computing the  $k$  smallest real numbers of the form  $a + b\sqrt{2}$  for nonnegative integers  $a$  and  $b$ .*

**Solution 10.5:** We can solve this problem using a min-heap  $H$  and a set  $S$  as follows. We initialize  $H$  to contain  $0 + 0\sqrt{2} = 0$ , and initialize  $S$  to the empty set. (A simple list will suffice to represent  $S$ ). We now iteratively do the following, stopping when  $S$  has  $k$  elements. When we perform an extract-min from  $H$  to obtain a number  $a + b\sqrt{2}$ ,

we add it to  $S$ , and compute  $c_1 = (a + 1) + b\sqrt{2}$  and  $c_2 = a + (b + 1)\sqrt{2}$  which we add to  $H$ .

Suppose for the sake of contradiction that  $S$  is not the desired set. Since  $|S| = k$ , there has to be at least one number in the desired set that is not in  $S$ . Let the smallest such number be  $m = p + q\sqrt{2}$ . Note that  $p$  and  $q$  cannot both be 0. Similarly, there must be a number  $l$  that is in  $S$  and is greater than all numbers in  $S_2^k$ . If  $p > 0$ , consider the number  $n = (p - 1) + q\sqrt{2}$ . It is less than  $m$ , and greater than 0, so it must be in  $S$ , since  $S$  contains all numbers in the desired set that are smaller than  $m$ . But then when we processed  $n$  to put it in  $S$ , we would have added  $n$  to  $H$ . This contradicts our adding  $l$  to  $S$ —the heap would always return  $n$  before  $l$ .

It is possible for a number to be inserted twice into the heap. For example, both  $1 + 2\sqrt{2}$  and  $2 + \sqrt{2}$  produce  $2 + 2\sqrt{2}$ . No number can be inserted more than twice: the irrationality of  $\sqrt{2}$  implies that  $a + b\sqrt{2} = c + d\sqrt{2}$  iff  $a = c$  and  $b = d$ . We can check for duplicates when we perform extract-min.

```

1 struct Num {
2     Num(int a, int b) : a(a), b(b), val(a + b * sqrt(2)) {}
3
4     bool operator<(const Num& n) const { return val > n.val; }
5
6     // Equal function for hash.
7     bool operator==(const Num& n) const { return a == n.a && b == n.b; }
8
9     int a, b;
10    double val;
11 };
12
13 // Hash function for Num.
14 struct HashNum {
15     size_t operator()(const Num& n) const {
16         return hash<int>()(n.a) ^ hash<int>()(n.b);
17     }
18 };
19
20 vector<Num> generate_first_k(int k) {
21     priority_queue<Num, vector<Num>> min_heap;
22     vector<Num> smallest;
23     unordered_set<Num, HashNum> hash;
24
25     // Initial for 0 + 0 * sqrt(2).
26     min_heap.emplace(0, 0);
27     hash.emplace(0, 0);
28
29     while (smallest.size() < k) {
30         Num s(min_heap.top());
31         smallest.emplace_back(s);
32         hash.erase(s);
33         min_heap.pop();
34
35         // Add the next two numbers derived from s.
36         Num c1(s.a + 1, s.b), c2(s.a, s.b + 1);
37         if (hash.emplace(c1).second) {

```



```

38     min_heap.emplace(c1);
39 }
40 if (hash.emplace(c2).second) {
41     min_heap.emplace(c2);
42 }
43 }
44 return smallest;
45 }

```

**Problem 11.1, pg. 68:** Write a method that takes a sorted array  $A$  and a key  $k$  and returns the index of the first occurrence of  $k$  in  $A$ . Return  $-1$  if  $k$  does not appear in  $A$ . For example, when applied to the array in Figure 11.1 on Page 68 your algorithm should return 3 if  $k = 108$ ; if  $k = 285$ , your algorithm should return 6.

**Solution 11.1:** The key idea is to search for  $k$ . However, even if we find  $k$ , after recording this we continue the search on the left subarray. The complexity bound is still  $O(\log n)$ —this is because each iteration reduces the size of the subarray being searched by half. In C++ code:

```

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 }

```

**$\epsilon$ -Variant 11.1.1:** Let  $A$  be an unsorted array of  $n$  integers, with  $A[0] \geq A[1]$  and  $A[n-2] \leq A[n-1]$ . Call an index  $i$  a *local minimum* if  $A[i]$  is less than or equal to its neighbors. How would you efficiently find a local minimum, if one exists?

**$\epsilon$ -Variant 11.1.2:** A sequence is said to be ascending if each element is greater than or equal to its predecessor; a descending sequence is one in which each element is less than or equal to its predecessor. A sequence is strictly ascending if each element is greater than its predecessor. Suppose it is known that an array  $A$  consists of an ascending sequence followed by a descending sequence. Design an algorithm for finding the maximum element in  $A$ . Solve the same problem when  $A$  consists of a strictly ascending sequence, followed by a descending sequence.

**Problem 11.2, pg. 68:** Design an algorithm that takes an abs-sorted array  $A$  and a number  $k$ , and returns a pair of indices of elements in  $A$  that sum up to  $k$ . For example, if the input to your algorithm is the array in Figure 11.2 on Page 68 and  $k = 167$ , your algorithm should output (3, 7). Output  $(-1, -1)$  if there is no such pair.

**Solution 11.2:** First consider the case where the array is sorted in the conventional sense. In this case we can start with the pair consisting of the first element and the last element:  $(A[0], A[n - 1])$ . Let  $s = A[0] + A[n - 1]$ . If  $s = k$ , we are done. If  $s < k$ , we increase the sum by moving to pair  $(A[1], A[n - 1])$ . We need never consider  $A[0]$ ; since the array is sorted, for all  $i$ ,  $A[0] + A[i] \leq A[0] + A[n - 1] = k < s$ . If  $s > k$ , we can decrease the sum by considering the pair  $(A[0], A[n - 2])$ ; by analogous reasoning, we need never consider  $A[n - 1]$  again. We iteratively continue this process till we have found a pair that sums up to  $k$  or the indices meet, in which case the search ends. This solution works in  $O(n)$  time and  $O(1)$  space in addition to the space needed to store  $A$ .

This approach will not work when the array entries are sorted by absolute value. In this instance, we need to consider three cases:

- (1.) Both the numbers in the pair are negative.
- (2.) Both the numbers in the pair are positive.
- (3.) One is negative and the other is positive.

For Cases (1.) and (2.), we can run the above algorithm separately by just limiting ourselves to either positive or negative numbers. For Case (3.), we can use the same approach where we have one index for positive numbers, one index for negative numbers, and they both start from the highest possible index and then go down.

```

1 pair<int, int> find_pair_sum_k(const vector<int>& A, int k) {
2     pair<int, int> ret = find_pos_neg_pair(A, k);
3     if (ret.first == -1 && ret.second == -1) {
4         return k >= 0 ? find_pair_using_comp(A, k, less<int>())
5                        : find_pair_using_comp(A, k, greater_equal<int>());
6     }
7     return ret;
8 }
9
10 template <typename Comp>
11 pair<int, int> find_pair_using_comp(const vector<int>& A, int k, Comp comp) {
12     pair<int, int> ret(0, A.size() - 1);
13     while (ret.first < ret.second && comp(A[ret.first], 0)) {
14         ++ret.first;
15     }
16     while (ret.first < ret.second && comp(A[ret.second], 0)) {
17         --ret.second;
18     }
19
20     while (ret.first < ret.second) {
21         if (A[ret.first] + A[ret.second] == k) {
22             return ret;
23         } else if (comp(A[ret.first] + A[ret.second], k)) {
24             do {
25                 ++ret.first;

```

```

26     } while (ret.first < ret.second && comp(A[ret.first], 0));
27   } else {
28     do {
29       --ret.second;
30     } while (ret.first < ret.second && comp(A[ret.second], 0));
31   }
32 }
33 return {-1, -1}; // no answer.
34 }
35
36 pair<int, int> find_pos_neg_pair(const vector<int>& A, int k) {
37   // ret.first for positive, and ret.second for negative.
38   pair<int, int> ret(A.size() - 1, A.size() - 1);
39   // Find the last positive or zero.
40   while (ret.first >= 0 && A[ret.first] < 0) {
41     --ret.first;
42   }
43
44   // Find the last negative.
45   while (ret.second >= 0 && A[ret.second] >= 0) {
46     --ret.second;
47   }
48
49   while (ret.first >= 0 && ret.second >= 0) {
50     if (A[ret.first] + A[ret.second] == k) {
51       return ret;
52     } else if (A[ret.first] + A[ret.second] > k) {
53       do {
54         --ret.first;
55       } while (ret.first >= 0 && A[ret.first] < 0);
56     } else { // A[ret.first] + A[ret.second] < k.
57       do {
58         --ret.second;
59       } while (ret.second >= 0 && A[ret.second] >= 0);
60     }
61   }
62   return {-1, -1}; // no answer.
63 }

```

A simpler solution is based on a hash table (Chapter 12) to store all the numbers and then for each number  $x$  in the array, look up  $k - x$  in the hash table. If the hash function does a good job of spreading the keys, the time complexity for this approach is  $O(n)$ . However, it requires  $O(n)$  additional storage. If the array is sorted on elements (and not absolute values), for each  $A[i]$  we can use binary search to find  $k - A[i]$ . This approach uses  $O(1)$  additional space and has time complexity  $O(n \log n)$ . However, it is strictly inferior to the two pointer technique described at the beginning of the solution.

**Variant 11.2.1:** Design an algorithm that takes as input an array of integers  $A$ , and an integer  $k$ , and returns a pair of indices  $i$  and  $j$  such that  $A[j] - A[i] = k$ , if such a pair exists.

**Problem 11.3, pg. 69:** Design an  $O(\log n)$  algorithm for finding the position of the smallest element in a cyclically sorted array. Assume all elements are distinct. For example, for the array in Figure 11.3 on Page 69, your algorithm should return 4.

**Solution 11.3:** 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  $I = [l, r]$  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  $A[m_I] > A[r]$  then  $[l_I, m_I]$  cannot contain the index of the minimum element. Therefore we can restrict the search to  $[m_I + 1, r]$ . If  $A[m_I] < A[r]$  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 }

```

Note that this problem cannot be solved in less than linear time when elements may be repeated. For example, if  $A$  consists of  $n - 1$  1s and a single 0, that 0 cannot be detected in the worst case without inspecting every element. Following is the C++ code for the scenario when elements may be repeated:

```

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

**Problem 11.4, pg. 69:** You are given two sorted arrays  $A$  and  $B$  of lengths  $m$  and  $n$ , respectively, and a positive integer  $k \in [1, m + n]$ . Design an algorithm that runs in  $O(\log k)$  time for computing the  $k$ -th smallest element in array formed by merging  $A$  and  $B$ . Array elements may be duplicated within and between  $A$  and  $B$ .

**Solution 11.4:** 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  $A[x] < B[k - 1 - x]$ , 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  $B[k - 1 - x]$ .

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  $A[x - 1] = B[k - 1 - x]$ , 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  $x$  lying outside the range of valid indices for  $B$ , i.e., outside  $[0, n - 1]$ . Setting  $b = \max(0, k - n)$  and  $t = \min(m, k)$  resolves this problem.

```

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

```

```

24 }
25
26 int A_b_1 = b <= 0 ? numeric_limits<int>::min() : A[b - 1];
27 int B_k_b_1 = k - b - 1 < 0 ? numeric_limits<int>::min() : B[k - b - 1];
28 return max(A_b_1, B_k_b_1);
29 }

```

**Problem 11.5, pg. 69:** Implement a function which takes as input a floating point variable  $x$  and returns  $\sqrt{x}$ .

**Solution 11.5:** One of the fastest ways to invert a fast-growing monotone function (such as the square function) is to do a binary search. Given  $x$ , we start with a lower bound  $l$  and an upper bound  $u$  on  $\sqrt{x}$ . We iteratively check if the square of midpoint  $m$  of  $[l, u]$  is smaller than, greater than, or equal to  $x$ . In the first case, we update the lower bound to  $m$ ; in the second case, we update the upper bound to  $m$ ; in the third case, we return  $m$ .

When checking for equality, we use a notion of tolerance, `eps`, since floating point arithmetic is not exact. This tolerance is user-specified.

Trivial choices for the initial lower bound and upper bound are 0 and the largest floating point number that is representable. If  $x \geq 1.0$ , we can tighten the lower and upper bounds to 1.0 and  $x$ , since  $x \geq 1.0 \Rightarrow x^2 \geq x$ . If  $x < 1.0 \Rightarrow x^2 < x$ , the previous choice of  $l$  and  $u$  is incorrect; instead, we can use  $x$  and 1.0. The time complexity is  $O(\log \frac{x}{\text{eps}})$  since the number of iterations is affected by the choice of `eps`. Care has to be taken to ensure the compare function is resilient to finite precision effects.

```

1 double square_root(double x) {
2     // Decide the search range according to x.
3     double l, r;
4     if (compare(x, 1.0) < 0) { // x < 1.0.
5         l = x, r = 1.0;
6     } else { // x >= 1.0.
7         l = 1.0, r = x;
8     }
9
10    // Keep searching if l < r.
11    while (compare(l, r) == -1) {
12        double m = l + 0.5 * (r - l);
13        double square_m = m * m;
14        if (compare(square_m, x) == 0) {
15            return m;
16        } else if (compare(square_m, x) == 1) {
17            r = m;
18        } else {
19            l = m;
20        }
21    }
22    return l;
23 }
24
25 // 0 means equal, -1 means smaller, and 1 means larger.
26 int compare(double a, double b) {

```

```

27 // Use normalization for precision problem.
28 double diff = (a - b) / b;
29 return diff < -numeric_limits<double>::epsilon()
30         ? -1
31        : diff > numeric_limits<double>::epsilon();
32 }

```

**Variant 11.5.1:** Given two positive floating point numbers  $x$  and  $y$ , how would you compute  $\frac{x}{y}$  to within a specified tolerance  $\epsilon$  if the division operator cannot be used? You cannot use any library functions, such as *log* and *exp*; addition and multiplication are acceptable.

**Problem 11.6, pg. 70:** Suppose you were given a file containing roughly one billion Internet Protocol (IP) addresses, each of which is a 32-bit unsigned integer. How would you programmatically find an IP address that is not in the file? Assume you have unlimited drive space but only two megabytes of RAM at your disposal.

**Solution 11.6:** In the first step, we build an array of  $2^{16}$  32-bit unsigned integers that is initialized to 0 and for every IP address in the file, we take its 16 most significant bits to index into this array and increment the count of that number. Since the file contains fewer than  $2^{32}$  numbers, there must be one entry in the array that is less than  $2^{16}$ . This tells us that there is at least one IP address which has those upper bits and is not in the file. In the second pass, we can focus only on the addresses that match this criterion and use a bit array of size  $2^{16}$  to identify one of the missing numbers.

```

1 int find_missing_element(istream* 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 }

```

```

27 }
28 }

```

**Problem 11.7, pg. 70:** You are reading a sequence of words from a very long stream. You know a priori that more than half the words are repetitions of a single word  $w$  (the “majority element”) but the positions where  $w$  occurs are unknown. Design an algorithm that makes a single pass over the stream and uses only a constant amount of memory to identify  $w$ .

**Solution 11.7:** The following observation leads to an elegant solution. If you take any two distinct elements from the stream and discard them away, the majority element remains the majority of the remaining elements. (This hinges on the assumption that there exists a majority element to begin with). The reasoning is follows.

**Proof:**

Let’s say the majority element occurred  $m$  times out of  $n$  elements in the stream such that  $\frac{m}{n} > \frac{1}{2}$ . The two distinct elements that are discarded can have at most one of the majority elements. Hence after discarding them, the ratio of the previously majority element to the total number of elements is either  $\frac{m}{(n-2)}$  or  $\frac{(m-1)}{(n-2)}$ . It is simple to verify that if  $\frac{m}{n} > \frac{1}{2}$ , then  $\frac{m}{(n-2)} > \frac{(m-1)}{(n-2)} > \frac{1}{2}$ .

Now, as we read the stream from beginning to the end, as soon as we encounter more than one distinct element, we can discard one instance of each element and what we are left with in the end must be the majority element.

```

1 string majority_search(istream* sin) {
2     string candidate, buf;
3     int count = 0;
4     while (*sin >> buf) {
5         if (count == 0) {
6             candidate = buf;
7             count = 1;
8         } else if (candidate == buf) {
9             ++count;
10        } else {
11            --count;
12        }
13    }
14    return candidate;
15 }

```

The code above assumes a majority word exists in the sequence. If no word has a strict majority, it still returns a word from the stream, albeit without any meaningful guarantees on how common that word is. We could check with a second pass whether the returned word was a majority.

**Problem 12.1, pg. 71:** Write a function that takes as input a dictionary of English words, and returns a partition of the dictionary into subsets of words that are all anagrams of each other.



**Solution 12.1:** Given a string  $s$ , let  $\text{sort}(s)$  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  $\text{sort}(x) = \text{sort}(y)$ . For example,  $\text{sort}(\text{"logarithmic"})$  and  $\text{sort}(\text{"algorithmic"})$  are both  $\text{"acghilmort"}$ . Therefore anagrams can be identified by adding  $\text{sort}(s)$  for each string  $s$  in the dictionary to a hash table.

```

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             copy(p.second.begin(),
16                 p.second.end(),
17                 ostream_iterator<string>(cout, " "));
18             cout << endl;
19         }
20     }
21 }

```

**Problem 12.2, pg. 72:** You are required to write a method which takes an anonymous letter  $L$  and text from a magazine  $M$ . Your method is to return **true** iff  $L$  can be written using  $M$ , i.e., if a letter appears  $k$  times in  $L$ , it must appear at least  $k$  times in  $M$ .

**Solution 12.2:** In the problem scenario, it is likely that the string encoding the magazine is much longer than the string encoding the anonymous letter. We build a hash table  $H_L$  for  $L$ , where each key is a character in the letter and its value is the number of times it appears in the letter. Consequently, we scan the magazine character-by-character. When processing  $c$ , if  $c$  appears in  $H_L$ , we reduce its frequency count by 1; we remove it from  $H_L$  when its count goes to zero. If  $H_L$  becomes empty, we return true. If it is nonempty when we get to the end of  $M$ , we return false.

```

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 }

```

**Remark:** If the characters are coded in ASCII, we could do away with  $H_L$  and use a 256 entry integer array  $A$ , with  $A[i]$  being set to the number of times the character  $i$  appears in the letter.

**Problem 12.3, pg. 72:** Let  $P$  be a set of  $n$  points in the plane. Each point has integer coordinates. Design an efficient algorithm for computing a line that contains the maximum number of points in  $P$ .

**Solution 12.3:** 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  $\frac{n(n-1)}{2}$  pairs of points, and for each pair we have to do a lookup in  $H$ , an insert into  $H$  if the defined line is not already in  $H$ , and two inserts into the corresponding set of points. The hash table operations are  $O(1)$  time, leading to an  $O(n^2)$  time bound for this part of the computation.

We finish by finding the line with the maximum number of points with a simple iteration through the hash table searching for the line with the most points in its corresponding set.

The design of a hash function appropriate for lines is more challenging than it may seem at first. The equation of line through  $(x_1, y_1)$  and  $(x_2, y_2)$  is

$$y = \frac{y_2 - y_1}{x_2 - x_1}x + \frac{x_2y_1 - x_1y_2}{x_2 - x_1}.$$

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.

```

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 13.1, pg. 73:** Sort lines of a text file that has one million lines such that the average length of a line is 100 characters but the longest line is one million characters long.

**Solution 13.1:** Almost all sorting algorithms rely on swapping records. However this becomes complicated when the record size varies. One way of dealing with this problem is to allocate for the maximum possible size for each record—this can be wasteful if there is a large variation in the sizes.

The better solution is *indirect sort*. First, build an array *P* of pointers to the records. Then sort the pointers using the compare function on the dereferenced pointers. Finally, iterate through *P* writing the dereferenced pointers.

```

1  void indirect_sort(const string& file_name) {
2      // Store file records into A.
3      ifstream ifs(file_name.c_str());
4      vector<int> A;
5      int x;
6      while (ifs >> x) {
7          A.emplace_back(x);
8      }
9
10     // Initialize P.
11     vector<const int*> P;
12     for (int& a : A) {
13         P.emplace_back(&a);
14     }
15
16     // Indirectly sort file.
17     sort(P.begin(), P.end(), [](const int * a, const int * b)->bool {
18         return *a < *b;
19     });
20
21     // Output file.
22     ofstream ofs(file_name.c_str());
23     for (const int* p : P) {
24         ofs << *p << endl;
25     }
26 }

```

**Problem 13.2, pg. 74:** You are given an array of *n* *Person* objects. Each *Person* object has a field *key*. Rearrange the elements of the array so that *Person* objects with equal keys

appear together. The order in which distinct keys appear is not important. Your algorithm must run in  $O(n)$  time and  $O(k)$  additional space. How would your solution change if keys have to appear in sorted order?

**Solution 13.2:** We use the approach described in the introduction to the problem. However, we cannot apply it directly, since we need to write objects, not integers—two objects may have the same key but other fields may be different.

We use a hash table  $C$  to count the number of distinct occurrences of each key. We iterate over each key  $k$  in  $C$  and keep a cumulative count  $s$  which is the starting offset in the array where elements with key  $k$  are to be placed. We put the key-value pair  $(k, s)$  in a hash table  $M$ —basically  $M$  partitions the array into the subarrays holding objects with equal keys.

We then iteratively get a key  $k$  from  $M$  and swap the element  $e$  at  $k$ 's current offset (which we get from  $M$ ) with the location appropriate for  $e$ 's key  $e.key$  (which we also get from  $M$ ). Since  $e$  is now in its correct location, we update  $M$  by advancing the offset corresponding to  $e.key$ , taking care to remove  $e.key$  from  $M$  when all elements with key equal to  $e.key$  are correctly placed.

The time complexity is  $O(n)$ , since the first pass entails  $n$  hash table inserts, and the second pass performs a constant amount of work to move one element to the right location. (Selecting an arbitrary key from a hash table is a constant time operation.) The additional space complexity dictated by  $C$  and  $M$ , and is  $O(k)$ , where  $k$  is the number of distinct keys.

```

1 struct Person {
2     bool operator<(const Person& that) const { return key < that.key; }
3
4     bool operator==(const Person& that) const { return key == that.key; }
5
6     bool operator!=(const Person& that) const { return key != that.key; }
7
8     int key;
9     string name;
10 };
11
12 // Hash function for Person.
13 struct HashPerson {
14     size_t operator()(const Person& n) const {
15         return hash<int>()(n.key) ^ hash<string>()(n.name);
16     }
17 };
18
19 void counting_sort(vector<Person>* people) {
20     unordered_map<int, int> key_to_count;
21     for (const Person& p : *people) {
22         ++key_to_count[p.key];
23     }
24     unordered_map<int, int> key_to_offset;
25     int offset = 0;
26     for (const auto& p : key_to_count) {
27         key_to_offset[p.first] = offset;
28         offset += p.second;

```

```

29 }
30
31 while (key_to_offset.size()) {
32     auto from = key_to_offset.begin();
33     auto to = key_to_offset.find((*people)[from->second].key);
34     swap((*people)[from->second], (*people)[to->second]);
35     // Use key_to_count to see when we are finished with a particular key.
36     if (--key_to_count[to->first]) {
37         ++to->second;
38     } else {
39         key_to_offset.erase(to);
40     }
41 }
42 }

```

If the objects are additionally required to appear in sorted key order, we can store  $M$  using a BST-based map instead of a hash table. The time complexity becomes  $O(n + k \log k)$ , since BST insertion takes time  $O(\log k)$ . This should make sense, since if  $k = n$ , we are doing a regular sort, which is known to be  $O(n \log n)$  for sorting based on comparisons.

**Problem 13.3, pg. 74:** Given sorted arrays  $A$  and  $B$  of lengths  $n$  and  $m$  respectively, return an array  $C$  containing elements common to  $A$  and  $B$ . The array  $C$  should be free of duplicates. How would you perform this intersection if—(1.)  $n \approx m$  and (2.)  $n \ll m$ ?

**Solution 13.3:** The simplest algorithm is a “loop join”, i.e., walking through all the elements of one array and comparing them to the elements of the other array. This has  $O(mn)$  time complexity, regardless of whether the arrays are sorted or unsorted:

```

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 }

```

Now our algorithm time complexity is  $O(n \log m)$ . We can further improve our run time by choosing the longer array for the inner loop since if  $n \ll m$  then  $m \log(n) \gg n \log(m)$ .

This is the best solution if one set is much smaller than the other. However it is not optimal for cases where the set sizes are similar because we are not using the fact that both arrays are sorted to our advantage. In that case, iterating in tandem through the elements of each array in increasing order will work best as shown in this C++ code:

```

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 13.4, pg. 74:** Given a set of events, how would you determine the maximum number of events that take place concurrently?

**Solution 13.4:** 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  $\tau$  is not the starting or ending time of an event.

This leads to an  $O(n^2)$  brute-force algorithm, where  $n$  is the number of intervals: for each interval, for each of its two endpoints, determining how many intervals contain that point. The total number of endpoints is  $2n$  and each check takes  $O(n)$  time, since checking whether an interval  $[b_i, e_i]$  contains a point  $t$  takes  $O(1)$  time (simply check if  $b_i \leq t \leq e_i$ ).

We can improve the run time to  $O(n \log n)$  by sorting the set of all the endpoints in ascending order. If two endpoints have equal times, and one is a start time and the other is an end time, the one corresponding to a start time comes first. (If both are start or finish times, we break ties arbitrarily.)

We initialize a counter to 0, and iterate through the sorted sequence  $S$  from smallest to largest. For each endpoint that is the start of an interval, we increment the counter by 1, and for each endpoint that is the end of an interval, we decrement the counter by 1. The maximum value attained by the counter is maximum number of overlapping intervals.

```

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 }

```

**$\epsilon$ -Variant 13.4.1:** Users  $1, 2, \dots, n$  share an Internet connection. User  $i$  uses  $b_i$  bandwidth from time  $s_i$  to  $f_i$ , inclusive. What is the peak bandwidth usage?

**Problem 13.5, pg. 75:** Design an algorithm that takes as input a set of intervals  $I$ , and outputs the union of the intervals. What is the time complexity of your algorithm as a function of the number of intervals?

**Solution 13.5:** We begin with by sorting the intervals  $I$  on their left endpoints. If left endpoints  $a$  and  $b$  are equal, with  $a$  corresponding to a closed interval and  $b$  to an open interval,  $a$  comes first; otherwise, we break ties arbitrarily.



Let the sorted sequence be  $\langle I_0, I_1, \dots, I_{n-1} \rangle$ . We create the result  $\langle R_0, R_1, \dots, R_m \rangle$  where  $m \leq n$  by processing intervals in order; the  $R_i$ s will be sorted by their left endpoints. Let  $t$  and  $s$  be interval-valued variables initialized to  $I_0$ , and  $I_1$ , respectively; we will show how to extend  $t$  to  $R_0$ . Let the left and right endpoints of  $t(s)$  be  $t.l(s.l)$  and  $t.r(s.r)$ , respectively. We have the following cases:

- $(s.l > t.r)$ :  $R_0$  is  $t$ , since no later interval can overlap or be adjacent to  $t$ .
- $(s.l = t.r)$  and  $(s$  is left open and  $t$  is right open): we set  $R_0$  to  $t$ , since  $s$  and  $t$  cannot be merged and no later interval can overlap or be adjacent to  $t$ .
- $(s.l < t.r)$  or  $(s.l = t.r$  and  $(s$  is left-closed or  $t$  is right-closed)): if  $s.r > t.r$  or  $(s.r = t.r$  and  $s$  is right-closed) we extend  $t$ 's right endpoint to  $s.r$ , and  $t$  is right open iff  $s$  is right open. We assign  $s$  to the next unprocessed interval and continue.

The code below implements this case analysis iteratively:

```

1 struct Interval {
2     private:
3         struct Endpoint {
4             bool isClose;
5             int val;
6         };
7
8     public:
9         bool operator<(const Interval& i) const {
10             return left.val != i.left.val ? left.val < i.left.val
11                 : (left.isClose && !i.left.isClose);
12         }
13
14         Endpoint left, right;
15 };
16
17 vector<Interval> Union_intervals(vector<Interval> I) {
18     // Empty input.
19     if (I.empty()) {
20         return {};
21     }
22
23     // Sort intervals according to their left endpoints.
24     sort(I.begin(), I.end());
25     Interval curr(I.front());
26     vector<Interval> uni;
27     for (int i = 1; i < I.size(); ++i) {
28         if (I[i].left.val < curr.right.val ||
29             (I[i].left.val == curr.right.val &&
30              (I[i].left.isClose || curr.right.isClose))) {
31             if (I[i].right.val > curr.right.val ||
32                 (I[i].right.val == curr.right.val && I[i].right.isClose)) {
33                 curr.right = I[i].right;
34             }
35         } else {
36             uni.emplace_back(curr);
37             curr = I[i];
38         }
39     }

```

```

39     }
40     uni.emplace_back(curr);
41     return uni;
42 }

```

**Problem 13.6, pg. 75:** Design an algorithm that takes as input an array  $A$  and a number  $t$ , and determines if  $A$  3-creates  $t$ .

**Solution 13.6:** We consider the case where  $k = 2$  and  $A$  is sorted in Problem 11.2 on Page 68. Therefore, one solution is to sort  $A$  and for each  $A[i]$ , search for indices  $j$  and  $k$  such that  $A[j] + A[k] = t - A[i]$ . The additional space needed is  $O(1)$ , and the time complexity is the sum of the time taken to sort,  $O(n \log n)$ , and then to run the  $O(n)$  algorithm in Solution 11.2 on Page 142  $n$  times, 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 (const 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 }

```

**Remark:** Surprisingly, it is possible, in theory, to improve the time complexity when the entries in  $A$  are nonnegative integers in a small range, specifically, the maximum entry is  $O(n)$ . The idea is to determine all possible 3-sums by encoding the array as a polynomial  $P_A(x) = \sum_{i=0}^{n-1} x^{A[i]}$ . The powers of  $x$  that appear in the polynomial  $P_A(x) \times P_A(x)$  corresponds to sums of pairs of elements in  $A$ ; similarly, the powers of  $x$  in  $P_A(x) \times P_A(x) \times P_A(x)$  correspond to sums of triples of elements in  $A$ . Two  $n$ -degree polynomials can be multiplied in  $O(n \log n)$  time using the fast Fourier Transform (FFT). The details are long and tedious, and the approach is unlikely to do well in practice.

**$\epsilon$ -Variant 13.6.1:** Solve the same problem when the three elements must be distinct. For example, if  $A = [5, 2, 3, 4, 3]$  and  $t = 9$ , then  $A[2] + A[2] + A[2]$  is not acceptable,  $A[2] + A[2] + A[4]$  is not acceptable, but  $A[1] + A[2] + A[3]$  and  $A[1] + A[3] + A[4]$  are acceptable.

**Variant 13.6.2:** Solve the same problem when  $k$  is an additional input.

**Problem 14.1, pg. 76:** Write a function that takes as input the root of a binary tree whose nodes have a key field, and returns `true` iff the tree satisfies the BST property.

**Solution 14.1:** Several solutions exist, which differ in terms of their space and time complexity, and the effort needed to code them.

The simplest is to start with the root  $r$ , and compute the maximum key  $r.left.max$  stored in the root's left subtree, and the minimum key  $r.right.min$  in the root's right subtree. Then we check that the key at the root is greater than or equal to  $r.right.min$  and less than or equal to  $r.left.max$ . If these checks pass, we continue checking the root's left and right subtree recursively.

Computing the minimum key in a binary tree is straightforward: we compare the key stored at the root with the minimum key stored in its left subtree and with the minimum key stored in its right subtree. The maximum key is computed similarly. (Note that the minimum may be in either subtree, since the tree may not satisfy the BST property.)

The problem with this approach is that it will repeatedly traverse subtrees. In a worst case, when the tree is BST and each node's left child is empty, its complexity is  $O(n^2)$ , where  $n$  is the number of nodes. The complexity can be improved to  $O(n)$  by caching the largest and smallest keys at each node; this requires  $O(n)$  additional storage.

We now present two approaches which have  $O(n)$  time complexity and  $O(h)$  additional space complexity.

The first, more straightforward approach, is to check constraints on the values for each subtree. The initial constraint comes from the root. Each node in its left (right) child must have a value less than or equal (greater than or equal) to the value at the root. This idea generalizes: if all nodes in a tree rooted at  $t$  must have values in the range  $[l, u]$ , and the value at  $t$  is  $w \in [l, u]$ , then all values in the left subtree of  $t$  must be in the range  $[l, w]$ , and all values stored in the right subtree of  $t$  must be in the range  $[w, u]$ . The code below uses this approach.

```

1  template <typename T>
2  bool is_BST(const unique_ptr<BinaryTree<T>>& r) {
3      return is_BST_helper(r, numeric_limits<T>::min(), numeric_limits<T>::max());
4  }
5
6  template <typename T>
7  bool is_BST_helper(const unique_ptr<BinaryTree<T>>& r,
8                    const T& lower,
9                    const T& upper) {
10     if (!r) {

```

```

11     return true;
12 } else if (r->data < lower || r->data > upper) {
13     return false;
14 }
15
16 return is_BST_helper(r->left, lower, r->data) &&
17        is_BST_helper(r->right, r->data, upper);
18 }

```

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  template <typename T>
2  bool is_BST(const unique_ptr<BinaryTree<T>>& root) {
3      auto* n = root.get();
4      // Store the value of previous visited node.
5      int last = numeric_limits<T>::min();
6      bool res = true;
7
8      while (n) {
9          if (n->left.get()) {
10             // Find the predecessor of n.
11             auto* pre = n->left.get();
12             while (pre->right.get() && pre->right.get() != n) {
13                 pre = pre->right.get();
14             }
15
16             // Process the successor link.
17             if (pre->right.get()) { // pre->right == n.
18                 // Revert the successor link if predecessor's successor is n.
19                 pre->right.release();
20                 if (last > n->data) {
21                     res = false;
22                 }
23                 last = n->data;
24                 n = n->right.get();
25             } else { // if predecessor's successor is not n.
26                 pre->right.reset(n);
27                 n = n->left.get();
28             }
29         } else {
30             if (last > n->data) {
31                 res = false;
32             }
33             last = n->data;
34             n = n->right.get();
35         }
36     }
37     return res;
38 }

```

The approaches outlined above all explore the left subtree first. Therefore, even if the BST property does not hold at a node which is close to the root (e.g., the key stored at the right child is less than the key stored at the root), their time complexity is still  $O(n)$ .

We can search for violations of the BST property in a BFS manner to reduce the time complexity when the property is violated at a node whose depth is small, specifically much less than  $n$ .

The code below uses a queue to process nodes. Each queue entry contains a node, as well as an upper and a lower bound on the keys stored at the subtree rooted at that node. The queue is initialized to the root, with lower bound  $-\infty$  and upper bound  $+\infty$ .

Suppose an entry with node  $n$ , lower bound  $l$  and upper bound  $u$  is popped. If  $n$ 's left child is not null, a new entry consisting of  $n.left$ , upper bound  $n.key$  and lower bound  $l$  is added. A symmetric entry is added if  $n$ 's right child is not null. When adding entries, we check that the node's key lies in the range specified by the lower bound and the upper bound; if not, we return immediately reporting a failure.

We claim that if the BST property is violated in the subtree consisting of nodes at depth  $d$  or less, it will be discovered without visiting any nodes at levels  $d + 1$  or more. This is because each time we enqueue an entry, the lower and upper bounds on the node's key are the tightest possible. A formal proof of this is by induction; intuitively, it is because we satisfy all the BST requirements induced by the search path to that node.

```

1  template <typename T>
2  struct QNode {
3      BinaryTree<T>* node;
4      T lower, upper;
5  };
6
7  template <typename T>
8  bool is_BST(const unique_ptr<BinaryTree<T>>& n) {
9      queue<QNode<T>> q;
10     q.emplace(
11         QNode<T>{n.get(), numeric_limits<T>::min(), numeric_limits<T>::max()});
12
13     while (!q.empty()) {
14         if (q.front().node) {
15             if (q.front().node->data < q.front().lower ||
16                 q.front().node->data > q.front().upper) {
17                 return false;
18             }
19
20             q.emplace(QNode<T>{q.front().node->left.get(), q.front().lower,
21                               q.front().node->data});
22             q.emplace(QNode<T>{q.front().node->right.get(), q.front().node->data,
23                               q.front().upper});
24         }
25         q.pop();
26     }
27     return true;

```

28 }

**Problem 14.2, pg. 77:** Write a function that takes a BST  $T$  and a key  $k$ , and returns the first entry larger than  $k$  that would appear in an inorder walk. If  $k$  is absent or no key larger than  $k$  is present, return `null`. For example, when applied to the BST in Figure 14.1 on Page 77 you should return 29 if  $k = 23$ ; if  $k = 32$ , you should return `null`.

**Solution 14.2:** 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  template <typename T>
2  BinarySearchTree<T>* find_first_larger_k_with_k_exist(
3      const unique_ptr<BinarySearchTree<T>>& r,
4      const T& k) {
5      bool found_k = false;
6      BinarySearchTree<T>* curr = r.get(), *first = nullptr;
7
8      while (curr) {
9          if (curr->data == k) {
10             found_k = true;
11             curr = curr->right.get();
12          } else if (curr->data > k) {
13             first = curr;
14             curr = curr->left.get();
15          } else { // curr->data < k.
16             curr = curr->right.get();
17          }
18      }
19      return found_k ? first : nullptr;
20 }

```

**Problem 14.3, pg. 77:** How would you build a BST of minimum possible height from a sorted array  $A$ ?

**Solution 14.3:** 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  template <typename T>
2  BinarySearchTree<T>* build_BST_from_sorted_array(const vector<T>& A) {
3      return build_BST_from_sorted_array_helper(A, 0, A.size());
4  }
5

```

```

6 // Build BST based on subarray A[start : end - 1].
7 template <typename T>
8 BinarySearchTree<T>* build_BST_from_sorted_array_helper(const vector<T>& A,
9                                                         int start,
10                                                         int end) {
11     if (start < end) {
12         int mid = start + ((end - start) >> 1);
13         return new BinarySearchTree<T>{
14             A[mid], unique_ptr<BinarySearchTree<T>>(
15                 build_BST_from_sorted_array_helper(A, start, mid)),
16             unique_ptr<BinarySearchTree<T>>(
17                 build_BST_from_sorted_array_helper(A, mid + 1, end))};
18     }
19     return nullptr;
20 }

```

**Problem 15.1, pg. 79:** Design an efficient algorithm to compute the diameter of a tree.

**Solution 15.1:** We can compute the diameter by running BFS, described on Page 87, from each node and recording the maximum value of the shortest path distances computed. This has  $O(|V|(|V| + |E|)) = O(|V|^2)$  time complexity since  $|E| = |V| - 1$  in a tree.

We can achieve better time complexity by using divide and conquer. First we define some notation. If  $T$  is a nonempty tree, let  $\text{root}(T)$  denote the node at the root of  $T$ . Let  $l_{u,v}$  be the length of the edge  $(u, v)$ . The *degree* of a node  $u$  in a rooted tree is the number of its children, denoted as  $m$ . Define the *weighted height*  $h_u$  of a tree rooted at  $u$  to be 0 if  $u$  is a leaf and  $\max_{1 \leq i \leq m} (l_{u, \text{root}(T_i)} + h_{\text{root}(T_i)})$ , where  $T_1, T_2, \dots, T_m$  are the subtrees rooted at  $u$ 's children.

Let  $T$  be a tree whose root is  $r$ . Suppose  $r$  has degree  $m$ . For now, assume  $m \geq 2$ . Let  $d_1, d_2, \dots, d_m$  be the diameters and  $h_1, h_2, \dots, h_m$  the weighted heights of the subtrees.

Let  $\lambda$  be a longest path in  $T$ . Either it passes through  $r$  or it does not. If  $\lambda$  does not pass through  $r$ , it must be entirely within one of the  $m$  subtrees and hence the longest path length in  $T$  is the maximum of  $d_1, d_2, \dots, d_m$ . If it does pass through  $r$ , it must be between a pair of nodes in distinct subtrees that are farthest from  $r$ . The distance from  $r$  to the node in  $T_i$  that is farthest from it is simply  $f_i = h_i + l_{r,i}$ . Therefore the longest length path in  $T$  is the larger of the maximum of  $d_1, d_2, \dots, d_m$  and the sum of the two largest  $f_i$ s.

Now we consider the cases  $m = 0$  and  $m = 1$ . If  $m = 0$  the subtree rooted at  $t$  is just the node  $t$  and the length of the longest path is 0. If  $m = 1$  the length of the longest path in  $t$  is  $\max(h_1 + l_{r,1}, d_1)$ .

The following algorithm computes the tree diameter. Process the tree in bottom-up fashion. For each node we process its subtrees one at a time. We update the maximum tree diameter based on the subtree weighted heights, diameters, and edge weights, using the observations above. The time complexity is proportional to the size of the tree, i.e.,  $O(|V|)$ .

```

1 struct TreeNode {
2     vector<pair<unique_ptr<TreeNode>, double>> edges;

```

```

3 };
4
5 double compute_diameter(const unique_ptr<TreeNode>& T) {
6     return T ? compute_height_and_diameter(T).second : 0.0;
7 }
8
9 // Return (height, diameter) pair.
10 pair<double, double> compute_height_and_diameter(
11     const unique_ptr<TreeNode>& r) {
12     double diameter = numeric_limits<double>::min();
13     array<double, 2> height = {{0.0, 0.0}}; // store the max two heights.
14     for (const auto& e : r->edges) {
15         pair<double, double> h_d = compute_height_and_diameter(e.first);
16         if (h_d.first + e.second > height[0]) {
17             height[1] = height[0];
18             height[0] = h_d.first + e.second;
19         } else if (h_d.first + e.second > height[1]) {
20             height[1] = h_d.first + e.second;
21         }
22         diameter = max(diameter, h_d.second);
23     }
24     return {height[0], max(diameter, height[0] + height[1])};
25 }

```

**Problem 15.2, pg. 82:** Given an array  $A$  of  $n$  numbers, find a longest subsequence  $\langle i_0, \dots, i_{k-1} \rangle$  such that  $i_j < i_{j+1}$  and  $A[i_j] \leq A[i_{j+1}]$  for any  $j \in [0, k-2]$ .

**Solution 15.2:** We present two solutions, an  $O(n^2)$ , and an  $O(n \log n)$  one.

We first describe the  $O(n^2)$  solution. Let  $s_i$  be the length of the longest nondecreasing subsequence of  $A$  that ends at  $A[i]$  (specifically,  $A[i]$  is included in this subsequence). Then we can write the following recurrence:

$$s_i = \max_{j \in [0, i-1]} \begin{pmatrix} s_j + 1, & \text{if } A[j] \leq A[i]; \\ 1, & \text{otherwise.} \end{pmatrix}$$

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:

```

1 vector<int> longest_nondecreasing_subsequence(const vector<int>& A) {
2     // Empty array.
3     if (A.empty() == true) {
4         return A;
5     }
6
7     vector<int> longest_length(A.size(), 1), previous_index(A.size(), -1);
8     int max_length_idx = 0;
9     for (int i = 1; i < A.size(); ++i) {
10         for (int j = 0; j < i; ++j) {
11             if (A[i] >= A[j] && longest_length[j] + 1 > longest_length[i]) {
12                 longest_length[i] = longest_length[j] + 1;

```



```

13     previous_index[i] = j;
14 }
15 }
16 // Record the index where longest subsequence ends.
17 if (longest_length[i] > longest_length[max_length_idx]) {
18     max_length_idx = i;
19 }
20 }
21
22 // Build the longest nondecreasing subsequence.
23 int max_length = longest_length[max_length_idx];
24 vector<int> ret(max_length);
25 while (max_length > 0) {
26     ret[--max_length] = A[max_length_idx];
27     max_length_idx = previous_index[max_length_idx];
28 }
29 return ret;
30 }

```

We now describe a subtler algorithm that has  $O(n \log n)$  complexity. Let  $M_{i,j}$  be the smallest possible tail value for any nondecreasing subsequence of length  $j$  using array elements  $A[0], A[1], \dots, A[i]$ . Note that for any  $i$ , we must have  $M_{i,1} \leq M_{i,2} \leq \dots \leq M_{i,j}$ .

We process  $A$ 's elements iteratively. When processing  $A[i + 1]$ , we look for the largest  $j$  such that  $M_{i,j} \leq A[i + 1]$ . 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  $A[0], A[1], \dots, A[i]$ , 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 }

```

**$\epsilon$ -Variant 15.2.1:** Define a sequence of numbers  $\langle a_0, a_1, \dots, a_{n-1} \rangle$  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  $\langle i_0, \dots, i_{k-1} \rangle$  such that  $\langle A[i_0], A[i_1], \dots, A[i_{k-1}] \rangle$  is alternating.

**$\epsilon$ -Variant 15.2.2:** Define a sequence of numbers  $\langle a_0, a_1, \dots, a_{n-1} \rangle$  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  $\langle i_0, \dots, i_{k-1} \rangle$  such that  $\langle A[i_0], A[i_1], \dots, A[i_{k-1}] \rangle$  is weakly alternating.

**$\epsilon$ -Variant 15.2.3:** Define a sequence of numbers  $\langle a_0, a_1, \dots, a_{n-1} \rangle$  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  $\langle i_0, \dots, i_{k-1} \rangle$  such that  $\langle A[i_0], A[i_1], \dots, A[i_{k-1}] \rangle$  is convex.

**$\epsilon$ -Variant 15.2.4:** Define a sequence of numbers  $\langle a_0, a_1, \dots, a_{n-1} \rangle$  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  $\langle i_0, \dots, i_{k-1} \rangle$  such that  $\langle A[i_0], A[i_1], \dots, A[i_{k-1}] \rangle$  is bitonic.

**$\epsilon$ -Variant 15.2.5:** Define a sequence of points in the plane to be *ascending* if each point is above and to the right of the previous point. How would you find a maximum ascending subset of a set of points in the plane?

**Problem 15.3, pg. 82:** Given two strings, represented as arrays of characters  $A$  and  $B$ , compute the minimum number of edits needed to transform the first string into the second string.

**Solution 15.3:** Let the Levenshtein distance between the two strings  $A$  and  $B$  be represented by  $E(A, B)$ . Let's say that  $a$  and  $b$  are, respectively, the length of strings  $A$  and  $B$ . We now make two claims:

- If  $A[a-1] = B[b-1]$ , i.e., the last character of  $A$  and  $B$  are the same, then  $E(A, B) = E(A[0 : a-2], B[0 : b-2])$ . This is because  $E(A[0 : a-2], B[0 : b-2])$  is an upper bound and a lower bound on  $E(A, B)$ . It is an upper bound, since one way to transform  $A$  to  $B$  is to transform  $A[0 : a-2]$  to  $B[0 : b-2]$ . It is a lower bound since we can take a transformation of  $A$  to  $B$ , and reorder the operations in it to get a transformation of  $A[0 : a-2]$  into  $B[0 : b-2]$  that is no longer than the original transformation.
- If  $A[a-1] \neq B[b-1]$ , i.e., the last two characters of the strings do not match, then

$$E(A, B) = 1 + \min \begin{pmatrix} E(A[0 : a-2], B[0 : b-2]), \\ E(A[0 : a-2], B), \\ E(A, B[0 : b-2]) \end{pmatrix}$$

Clearly, the expression on the right hand side is an upper bound on  $E(A, B)$ . To show that it is a lower bound, if a smaller sequence transforms  $A$  into  $B$ , there must be a step where the last character of  $A$  becomes the same as the last character of  $B$ . This could happen either by inserting a new character at the end, deleting the last character, or substituting the last character of  $A$  with

the last character of  $B$ . We can reorder the sequence such that this operation happens at the end. The length of the sequence would remain the same and we would still end up with  $B$  in the end. If this operation was a “delete”, then by deleting this operation, we get a sequence of operations that turn  $A[0, a-2]$  into  $B$ . If this operation was an “insert”, then by dropping this operation, we would have a set of transformations that turn  $A$  into  $B[0, b-2]$ . If this operation was a “substitute”, then by discarding this operation, we would have a set of transformations that turn  $A[0, a-2]$  into  $B[0, b-2]$ . In any of those cases, it would be a contradiction if there was a sequence of operations that turned  $A$  into  $B$  which is smaller than  $\min(E(A[0 : a-2], B[0 : b-2]), E(A[0 : a-2], B), E(A, B[0 : b-2])) + 1$ .

We use the above claims to compute  $E(A, B)$ . Specifically, we tabulate the values of  $E(A[0 : k], B[0 : l])$  for all values of  $k < a$  and  $l < b$ . This takes  $O(ab)$  time. We can implement this algorithm using  $O(\min(a, b))$  space by reusing space, since we never need more than one row of prior solution at a time. Following is the code in C++.

```

1 int Levenshtein_distance(string A, string B) {
2     // Try to reduce the space usage.
3     if (A.size() < B.size()) {
4         swap(A, B);
5     }
6
7     vector<int> D(B.size() + 1);
8     // Initialization.
9     iota(D.begin(), D.end(), 0);
10
11     for (int i = 1; i <= A.size(); ++i) {
12         int pre_i_1_j_1 = D[0]; // stores the value of D[i - 1][j - 1].
13         D[0] = i;
14         for (int j = 1; j <= B.size(); ++j) {
15             int pre_i_1_j = D[j]; // stores the value of D[i - 1][j].
16             D[j] = A[i - 1] == B[j - 1] ? pre_i_1_j_1
17                 : 1 + min(pre_i_1_j_1, min(D[j - 1], D[j]));
18             // Previous D[i - 1][j] will become the next D[i - 1][j - 1].
19             pre_i_1_j_1 = pre_i_1_j;
20         }
21     }
22     return D.back();
23 }

```

Figure 22.3 on the following page shows the  $E$  values for the strings “Carthorse” and “Orchestra”. Upper-case and lower-case characters are treated as being different. The Levenshtein distance for this two strings is 8. The longest subsequence which is present in both strings is  $\langle r, h, s \rangle$ .

**e-Variant 15.3.1:** Given  $A$  and  $B$  as above, compute a longest sequence of characters that is a subsequence of  $A$  and of  $B$ .

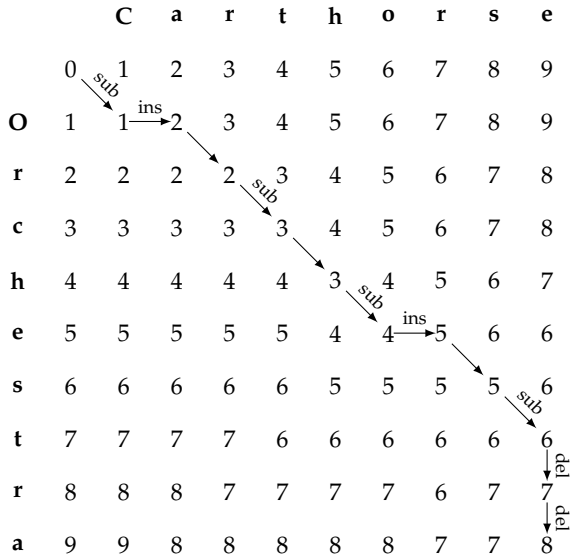


Figure 22.3: The  $E$  table for “Carthorse” and “Orchestra”.

**$\epsilon$ -Variant 15.3.2:** Given a string  $A$ , compute the minimum number of characters you need to delete from  $A$  to make the resulting string a palindrome.

**Variant 15.3.3:** Given a string  $A$  and a regular expression  $r$ , what is the string in the language of the regular expression  $r$  that is closest to  $A$ ? The distance between strings is the Levenshtein distance specified above.

**Problem 15.4, pg. 82:** Given a dictionary and a string  $s$ , design an efficient algorithm that checks whether  $s$  is the concatenation of a sequence of dictionary words. If such a concatenation exists, your algorithm should output it.

**Solution 15.4:** 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 in  $O(1)$  time. Then  $T[k]$  holds iff one of the following two conditions is true:

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

This tells us if we can break a given string into valid words, but does not yield the words themselves. We can obtain the words with a little more book-keeping. In table  $T$ , along with the Boolean value, we also store the length of the last word in the string.

```
1 vector<string> word_breaking(const string& s,  
2                             const unordered_set<string>& dict) {
```

```

3 // T[i] stores the length of the last string which composed 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[j] != 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 }

```

If we want all possible decompositions, we can store all possible values of  $j$  that gives us a correct break with each position. However the number of possible decompositions can be exponential here. This is exemplified by the string “itsitsitsits...”.

**Problem 15.5, pg. 82:** You have an aggregate score  $s$  and  $W$  which specifies the points that can be scored in an individual play. How would you find the number of combinations of plays that result in an aggregate score of  $s$ ? How would you compute the number of distinct sequences of individual plays that result in a score of  $s$ ?

**Solution 15.5:** Let  $W = \{w_0, w_1, \dots, w_{n-1}\}$  be the possible scores for individual plays. Let  $X$  be the set  $\{\langle x_0, x_1, \dots, x_{n-1} \rangle \mid \sum_{i=0}^{n-1} w_i x_i = s\}$ . 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  $\{\langle x_0, x_1, \dots, x_{n-1} \rangle\}$ , where  $0 \leq x_0 \leq \lfloor \frac{s}{w_0} \rfloor$ . 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  $\{w_1, w_2, \dots, w_{n-1}\}$ . The base case corresponds to computing the number of ways in which a score  $t \leq s$  can be formed with the  $w_{n-1}$ -score plays, which is 1 or 0, depending on whether  $w_{n-1}$  evenly divides  $t$ .

The algorithm outlined above has exponential complexity. We can use DP to reduce its complexity—for each  $t \leq s$  and  $d \in [1, n - 1]$  we cache the number of combinations of ways in which  $w_d, \dots, w_{n-1}$  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 }

```

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 }

```

**Variant 15.5.1:** Suppose the final score is given in the form  $(s, s')$ , i.e., Team 1 scored  $s$  points and Team 2 scored  $s'$  points. How would you compute the number of distinct scoring sequences which result in this score? For example, if the final score is  $(6, 3)$  then Team 1 scores 3, Team 2 scores 3, Team 1 scores 3 is a scoring sequence which results in this score.

**Variant 15.5.2:** Suppose the final score is  $(s, s')$ . How would you compute the maximum number of times the team that lead could have changed? For example, if  $s = 10$  and  $s' = 6$ , the lead could have changed 4 times: Team 1 scores 2, then Team 2 scores 3 (lead change), then Team 1 scores 2 (lead change), then Team 2 scores 3 (lead change), then Team 1 scores 3 (lead change) followed by 3.

**Problem 15.6, pg. 83:** How many ways can you go from the top-left to the bottom-right in an  $n \times m$  2D array? How would you count the number of ways in the presence of obstacles, specified by an  $n \times m$  Boolean 2D array  $B$ , where a **true** represents an obstacle.

**Solution 15.6:** This problem can be solved using a straightforward application of DP: the number of ways to get to  $(i, j)$  is the number of ways to get to  $(i - 1, j)$  plus the number of ways to get to  $(i, j - 1)$ . (If  $i = 0$  or  $j = 0$ , there is only one way to get to  $(i, j)$ .) The matrix storing the number of ways to get to  $(i, j)$  for the configuration in Figure 15.5 on Page 83 is shown in Figure 22.4.

```

1 int number_of_ways(int n, int m) {
2     vector<vector<int>> A(n, vector<int>(m, 0));
3     A[0][0] = 1; // one way to start from (0, 0).
4     for (int i = 0; i < n; ++i) {
5         for (int j = 0; j < m; ++j) {
6             A[i][j] += (i < 1 ? 0 : A[i - 1][j]) + (j < 1 ? 0 : A[i][j - 1]);
7         }
8     }
9     return A.back().back();
10 }

```

We can improve on the above by noting that we do not need an  $n \times m$  2D array, since to fill in the  $i$ -th row we do not need values from rows before  $i - 1$ .

1	1	1	1	1
1	2	3	4	5
1	3	6	10	15
1	4	10	20	35
1	5	15	35	70

**Figure 22.4:** The number of ways to get from  $(0, 0)$  to  $(i, j)$  for  $0 \leq i, j \leq 4$ .

An even better solution is based on the fact that each path from  $(0, 0)$  to  $(n - 1, m - 1)$  is a sequence of  $m - 1$  horizontal steps and  $n - 1$  vertical steps. There are  $\binom{n+m-2}{n-1} = \binom{n+m-2}{m-1} = \frac{(n+m-2)!}{(n-1)!(m-1)!}$  such paths.

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 }

```

```

17     }
18     return A.back().back();
19 }

```

**Variant 15.6.1:** A decimal number is a sequence of digits, i.e., a sequence over  $\{0, 1, 2, \dots, 9\}$ . The sequence has to be of length 1 or more, and the first element in the sequence cannot be 0. Call a decimal number  $D$  *monotone* if  $D[i] \leq D[i+1], 0 \leq i < |D|$ . Write a function which takes as input a positive integer  $k$  and computes the number of decimal numbers of length  $k$  that are monotone.

**Variant 15.6.2:** Call a decimal number  $D$ , as defined above, *strictly monotone* if  $D[i] < D[i+1], 0 \leq i < |D|$ . Write a function which takes as input a positive integer  $k$  and computes the number of decimal numbers of length  $k$  that are strictly monotone.

**Problem 15.7, pg. 84:** Given a set of symbols with corresponding frequencies, find a code book that has the smallest average code length.

**Solution 15.7:** 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 BinaryTree {
8     double prob;
9     Symbol* s;
10    shared_ptr<BinaryTree> left, right;
11 };
12
13 struct Compare {
14     bool operator()(const shared_ptr<BinaryTree>& lhs,
15                     const shared_ptr<BinaryTree>& rhs) const {
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<BinaryTree>,
23                   vector<shared_ptr<BinaryTree>>,
24                   Compare> min_heap;
25     for (auto& s : *symbols) {
26         min_heap.emplace(new BinaryTree{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<BinaryTree> l = min_heap.top();
32         min_heap.pop();
33         shared_ptr<BinaryTree> r = min_heap.top();
34         min_heap.pop();
35         min_heap.emplace(new BinaryTree{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<BinaryTree>& r, const string& s) {
44     if (r) {
45         // This node (i.e., leaf) contains symbol.
46         if (r->s) {
47             r->s->code = s;
48         } else { // non-leaf node.
49             assign_huffman_code(r->left, s + '0');
50             assign_huffman_code(r->right, s + '1');
51         }
52     }
53 }

```

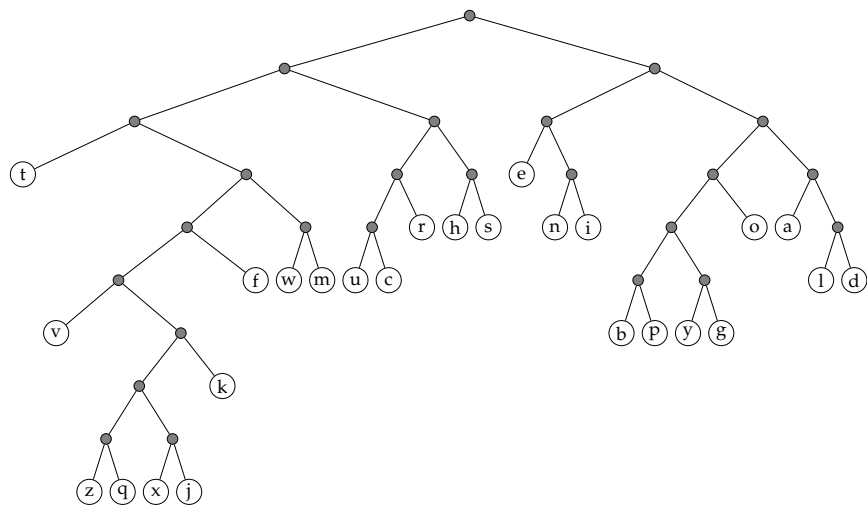
Applying this algorithm to the frequencies for English characters presented in Table 15.1 on Page 84 yields the Huffman tree in Figure 22.5 on the next page. The path from root to leaf yields that character's Huffman code, which is listed in Table 22.1 on the following page. For example, the codes for *t*, *e*, and *z* are 000, 100, and 001001000, respectively.

The codebook is explicitly given in Table 22.1 on the next page. The average code length for this coding is 4.205. In contrast, the trivial coding takes  $\lceil \log 26 \rceil = 5$  bits for each character.

Although it is unlikely that a rigorous proof of optimality would be asked in an interview setting, we give a proof by induction on the number of characters.

**Proof:**

For a single character, Huffman codes are trivially optimum. Let's say that for any distribution of frequencies among  $n$  characters, Huffman codes are optimum. Given this assumption, we will prove it is true for  $n + 1$  characters. We denote the



**Figure 22.5:** A Huffman tree for the English characters, assuming the frequencies given in Table 15.1 on Page 84.

**Table 22.1:** Huffman codes for English characters, assuming the frequencies given in Table 15.1 on Page 84.

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		

frequency of character  $c$  by  $f(c)$ .

Suppose there exists an encoding that has a smaller average length of code for some frequency distribution for  $n + 1$  characters.

For any encoding, we can map the codes to a binary tree by identifying the null string with root and adding a left edge for each 0 and a right edge for each 1.

We make several observations about a tree  $T$  corresponding to an optimum encoding:

- Each character must map to a leaf node; otherwise, the coding will violate our requirements on code prefixes.
- There cannot be a non-leaf node in  $T$  that has fewer than two children (otherwise, we can delete that node, bring its child one level up, and hence reduce the average code length).
- If we sort the leaves of  $T$  by their depths, the two deepest leaves must have the same depth (since the parent of the leaf with the longest path must have

another child).

- The two deepest leaves in  $T$  must be assigned to two characters with smallest frequencies (otherwise, we can swap characters and achieve smaller average code length).
- Suppose we remove the two smallest frequency characters  $s$  and  $t$  and replace them with a new character  $u$  that has its frequency equal to  $f(s) + f(t)$ . Then the optimum prefix coding for this set  $C'$  of characters and corresponding frequencies must have the same average code length as the tree  $T'$  that results from deleting the two lowest frequency leaves in  $T$  (which as previously argued can be taken as siblings) and assigning their parent's frequency to be the sum of these two frequencies. Otherwise, if a tree  $S'$  for  $C'$  has lower average code length than  $T'$ , we can create a tree  $S$  from  $S'$  by replacing the leaf corresponding to  $u$  with an internal node with two children corresponding to  $s$  and  $t$ . The average code length for the tree  $S$  is  $f(s) + f(t)$  plus the average code length of the tree  $S'$ , which, by hypothesis, has a lower average code length than  $T'$ . Since by construction, the average code length of  $T$  is  $f(s) + f(t)$  plus the average code length of  $T'$ , this contradicts the assumed optimality of  $T$ .

Now suppose the characters have frequencies  $p_1 \geq p_2 \geq \dots \geq p_{n+1}$ . Let  $A(p_1, \dots, p_{n+1})$  be the optimum average code length for this frequency distribution and  $\mathcal{H}(p_1, \dots, p_{n+1})$  be the average code length for Huffman coding.

From the above observations, it follows that

$$A(p_1, \dots, p_{n+1}) = A(p_1, \dots, p_{n-1}, p_n + p_{n+1}) + (p_n + p_{n+1}).$$

From the construction of Huffman codes we know that

$$\mathcal{H}(p_1, \dots, p_{n+1}) = \mathcal{H}(p_1, \dots, p_{n-1}, p_n + p_{n+1}) + (p_n + p_{n+1}).$$

By our inductive assumption,  $\mathcal{H}(p_1, \dots, p_{n-1}, p_n + p_{n+1}) = A(p_1, \dots, p_{n-1}, p_n + p_{n+1})$ . Hence  $\mathcal{H}(p_1, \dots, p_{n+1}) = A(p_1, \dots, p_{n+1})$ . In other words, Huffman coding is optimum for  $n + 1$  characters if it is optimum for  $n$  characters.

**Problem 16.1, pg. 87:** *Given a 2D array of black and white entries representing a maze with designated entrance and exit points, find a path from the entrance to the exit, if one exists.*

**Solution 16.1:** Model the maze as an undirected graph. Each vertex corresponds to a white pixel. We will index the vertices based on the coordinates of the corresponding pixel; so, vertex  $v_{i,j}$  corresponds to the 2D array entry  $(i, j)$ . Use edges to model adjacent pixels:  $v_{i,j}$  is connected to vertices  $v_{i+1,j}$ ,  $v_{i,j+1}$ ,  $v_{i-1,j}$ , and  $v_{i,j-1}$ , assuming these vertices exist—vertex  $v_{a,b}$  does not exist if the corresponding pixel is black or the coordinates  $(a, b)$  lie outside the image.

Now, run a DFS starting from the vertex corresponding to the entrance. If at some point, we discover the exit vertex in the DFS, then there exists a path from the

entrance to the exit. If we implement recursive DFS then the path would consist of all the vertices in the call stack corresponding to previous recursive calls to the DFS routine.

This problem can also be solved using BFS from the entrance vertex on the same graph model. The BFS tree has the property that the computed path will be a shortest path from the entrance. However BFS is more difficult to implement than DFS since in DFS, the compiler implicitly handles the DFS stack, whereas in BFS, the queue has to be explicitly coded. Since the problem did not call for a shortest path, it is better to use DFS.

```

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(&maze, s, e, 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(vector<vector<int>>* maze,
23                          const Coordinate& cur,
24                          const Coordinate& e,
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(maze, next, e, 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 }

```

**Problem 16.2, pg. 88:** Given a dictionary  $D$  and two strings  $s$  and  $t$ , write a function to determine if  $s$  produces  $t$ . Assume that all characters are lowercase alphabets. If  $s$  does produce  $t$ , output the length of a shortest production sequence; otherwise, output  $-1$ .

**Solution 16.2:** Define the undirected graph  $G = (D, E)$  by  $(u, v) \in E$  iff  $|u| = |v|$ , and  $u$  and  $v$  differ in one character. (Note that the relation “differs in one character” is symmetric, which is why the graph is undirected.)

A production sequence is simply a path in  $G$ , so what we need is a shortest path from  $s$  to  $t$  in  $G$ . Shortest paths in an undirected graph are naturally computed using BFS. We use a queue and a hash table of vertices (which indicates if a vertex has already been visited). We enumerate neighbors of a vertex  $v$  by an outer loop that iterates over each position in  $v$  and an inner loop that iterates over each choice of character for that position.

```

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 }

```

**Problem 16.3, pg. 89:** You are the photographer at a sporting event. You have to take pictures of teams. Each team has the same number of players. A team photo consist of rows of players. Each row consists of players from from one team. Each player must be taller than the player in front of him. Players within a row are equally spaced.

**Solution 16.3:** Let  $G$  be the DAG with vertices corresponding to the teams as follows and edges from vertex  $X$  to  $Y$  iff  $\text{sort}(X) < \text{sort}(Y)$ .

Every sequence of teams where a team can be placed behind its predecessor corresponds to a path in  $G$ . To find the longest such sequence, we simply need to find the longest path in the DAG  $G$ . We can do this, for example, by topologically ordering the vertices in  $G$ ; the longest path terminating at vertex  $v$  is the maximum of the longest paths terminating at  $v$ 's fan-ins concatenated with  $v$  itself.

The topological ordering computation is  $O(|V| + |E|)$  and dominates the computation time.

```

1 struct GraphVertex {
2     vector<GraphVertex*> edges;
3     int max_distance = 1;
4     bool visited = false;
5 };
6
7 int find_largest_number_teams(vector<GraphVertex*> G) {
8     stack<GraphVertex*> vertex_order(build_topological_ordering(G));
9     return find_longest_path(&vertex_order);
10 }
11
12 stack<GraphVertex*> build_topological_ordering(vector<GraphVertex*> G) {
13     stack<GraphVertex*> vertex_order;
14     for (auto& g : *G) {
15         if (!g.visited) {
16             DFS(&g, &vertex_order);
17         }
18     }
19     return vertex_order;
20 }
21
22 int find_longest_path(stack<GraphVertex*> vertex_order) {
23     int max_distance = 0;
24     while (!vertex_order->empty()) {
25         GraphVertex* u = vertex_order->top();
26         max_distance = max(max_distance, u->max_distance);
27         for (GraphVertex*& v : u->edges) {
28             v->max_distance = max(v->max_distance, u->max_distance + 1);
29         }
30         vertex_order->pop();
31     }
32     return max_distance;

```

```

33 }
34
35 void DFS(GraphVertex* cur, stack<GraphVertex*>* vertex_order) {
36     cur->visited = true;
37     for (const auto& next : cur->edges) {
38         if (!next->visited) {
39             DFS(next, vertex_order);
40         }
41     }
42     vertex_order->emplace(cur);
43 }

```

**Problem 16.4, pg. 89:** *Given an instance of the task scheduling problem, compute the least amount of time in which all the tasks can be performed, assuming an unlimited number of servers. Explicitly check that the system is feasible.*

**Solution 16.4:** 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  $\tau_i$  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 16.5, pg. 89:** *Design an algorithm which takes as input a graph  $G = (V, E)$ , directed or undirected, a nonnegative cost function on  $E$ , and vertices  $s$  and  $t$ ; your algorithm should output a path with the fewest edges amongst all shortest paths from  $s$  to  $t$ .*

**Solution 16.5:** Dijkstra's shortest path algorithm uses scalar values for edge length. However it can easily be modified to the case where the edge weight is a pair if *addition* and *comparison* can be defined over these pairs. In this case, if the edge cost is  $c$ , we say the length of the edge is given by the pair  $(c, 1)$ . We define addition to be just component-wise addition. Hence if we sum up the edge lengths over a path, we essentially get the total cost and the number of edges in the path. The compare function is lexicographic, first the total cost, then the number of edges. We can run Dijkstra's shortest path algorithm with this compare function and find the shortest path that requires the least number of edges.

Since a heap does not support efficient updates, it is more convenient to use a BST than a heap to implement the algorithm.

```

1 struct GraphVertex {
2     // distance stores (dis, #edges) pair.
3     pair<int, int> distance = {numeric_limits<int>::max(), 0};
4     vector<pair<GraphVertex*, int>> edges;
5     int id; // the id of this vertex.
6     GraphVertex* pred = nullptr; // the predecessor in the shortest path.
7 };
8
9 struct Comp {
10     bool operator()(const GraphVertex* lhs, const GraphVertex* rhs) const {
11         return lhs->distance.first < rhs->distance.first ||
12             (lhs->distance.first == rhs->distance.first &&
13              lhs->distance.second < rhs->distance.second);
14     }
15 };
16
17 void Dijkstra_shortest_path(GraphVertex* s, GraphVertex* t) {
18     // Initialization the distance of starting point.
19     s->distance = {0, 0};
20     set<GraphVertex*, Comp> node_set;
21     node_set.emplace(s);
22
23     while (!node_set.empty()) {
24         // Extract the minimum distance vertex from heap.
25         GraphVertex* u = *node_set.cbegin();
26         if (u == t) {
27             break;
28         }
29         node_set.erase(node_set.cbegin());
30
31         // Relax neighboring vertices of u.
32         for (const auto& v : u->edges) {
33             int v_distance = u->distance.first + v.second;
34             int v_num_edges = u->distance.second + 1;
35             if (v.first->distance.first > v_distance ||
36                 (v.first->distance.first == v_distance &&
37                  v.first->distance.second > v_num_edges)) {
38                 node_set.erase(v.first);
39                 v.first->pred = u;
40                 v.first->distance = {v_distance, v_num_edges};
41                 node_set.emplace(v.first);
42             }
43         }
44     }
45
46     // Output the shortest path with fewest edges.
47     output_shortest_path(t);
48 }
49
50 void output_shortest_path(GraphVertex*& v) {
51     if (v) {
52         output_shortest_path(v->pred);
53         cout << v->id << " ";
54     }

```



55 }

**$\epsilon$ -Variant 16.5.1:** Solve the same problem when edge weights are integers in  $(-\infty, \infty)$ . You may modify the graph, but must use an unmodified shortest path algorithm.

**Problem 17.1, pg. 92:** How would you programmatically determine if a tie is possible in a presidential election with two candidates, R and D?

**Solution 17.1:** We need to determine if there exists a subset of states whose Electoral College votes add up to  $\frac{538}{2} = 269$ . This is an instance of the subset sum problem, and is known to be NP-complete. It is a specialization of the 0-1 knapsack problem described in Problem 17.2 on Page 92 and the DP solution to that problem can be used. Following is the code in C++:

```

1 // V contains the number of votes for each state.
2 long ties_election(const vector<int>& V) {
3     int total_votes = accumulate(V.cbegin(), V.cend(), 0);
4
5     // No way to tie if the total number of votes is odd.
6     if (total_votes & 1) {
7         return 0;
8     }
9
10    vector<vector<long>> table(V.size() + 1, vector<long>(total_votes + 1, 0));
11    table[0][0] = 1; // base condition: 1 way to reach 0.
12    for (int i = 0; i < V.size(); ++i) {
13        for (int j = 0; j <= total_votes; ++j) {
14            table[i + 1][j] = table[i][j] + (j >= V[i] ? table[i][j - V[i]] : 0);
15        }
16    }
17    return table[V.size()][total_votes >> 1];
18 }

```

**Problem 17.2, pg. 92:** Design an algorithm for the knapsack problem that selects a subset of items that has maximum value and weighs at most  $w$  ounces. All items have integer weights and values.

**Solution 17.2:** Let  $V[i, w]$  be the maximum value that can be packed with weight less than or equal to  $w$  using the first  $i$  clocks. Then  $V[i, w]$  satisfies the following recurrence:

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

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. Note that the space complexity can be improved to  $O(w)$  by using a one-dimensional array to store the current optimal result and rewriting the next step result back to this array. Following is the code in C++:

```

1 int knapsack(int w, const vector<pair<int, int>>& items) {
2     vector<int> V(w + 1, 0);
3     for (int i = 0; i < items.size(); ++i) {
4         for (int j = w; j >= items[i].first; --j) {
5             V[j] = max(V[j], V[j - items[i].first] + items[i].second);
6         }
7     }
8     return V[w];
9 }

```

**Variant 17.2.1:** Solve the knapsack problem when the thief can take a fractional amount of an item.

**Problem 17.3, pg. 93:** Write a program that determines a sequence of steps by which the required amount of milk can be obtained using the worn-out jugs. The milk is being added to a large mixing bowl, and hence cannot be removed from the bowl. Furthermore, it is not possible to pour one jug's contents into another. Your scheme should always work, i.e., return between 2100 and 2300 mL of milk, independent of how much is chosen in each individual step, as long as that quantity satisfies the given constraints.

**Solution 17.3:** It is natural to solve this problem using recursion—if we use jug *A* for the last step, we need to correctly measure a volume of milk that is at least  $2100 - 230 = 1870$  mL—the last measurement may be as little as 230 mL, and anything less than 1870 mL runs the risk of being too little. Similarly, the volume must be at most  $2300 - 240 = 2060$  mL. The volume is not achievable if it is not achievable with any of the three jugs as ending points. We cache intermediate computations to reduce the number of recursive calls.

In the following code, we implement a general purpose function which finds the feasibility among *n* jugs; those arrays are passed in as *jugs*.

```

1 struct Jug {
2     int low, high;
3 };
4
5 class PairEqual {
6 public:
7     bool operator()(const pair<int, int>& a, const pair<int, int>& b) const {
8         return a.first == b.first && a.second == b.second;
9     }
10 };
11
12 struct HashPair {
13     size_t operator()(const pair<int, int>& p) const {
14         return hash<int>()(p.first) ^ hash<int>()(p.second);
15     }
16 };
17
18 bool check_feasible_helper(
19     const vector<Jug>& jugs,

```

```

20     int L,
21     int H,
22     unordered_set<pair<int, int>, HashPair, PairEqual>* c) {
23     if (L > H || c->find({L, H}) != c->cend() || (L < 0 && H < 0)) {
24         return false;
25     }
26
27     // Checks the volume for each jug to see if it is possible.
28     for (const Jug& j : jugs) {
29         if ((L <= j.low && j.high <= H) || // base case: j is contained in [L, H]
30             check_feasible_helper(jugs, L - j.low, H - j.high, c)) {
31             return true;
32         }
33     }
34     c->emplace(L, H); // marks this as impossible
35     return false;
36 }
37
38 bool check_feasible(const vector<Jug>& jugs, int L, int H) {
39     unordered_set<pair<int, int>, HashPair, PairEqual> cache;
40     return check_feasible_helper(jugs, L, H, &cache);
41 }

```

**Variant 17.3.1:** Suppose Jug  $i$  can be used to measure any quantity in  $[l_i, u_i]$  exactly. Determine if it is possible to measure a quantity of milk between  $L$  and  $U$ .

**Problem 17.4, pg. 93:** Implement a Sudoku solver. Your program should read an instance of Sudoku from the command line. The command line argument is a sequence of 3-digit strings, each encoding a row, a column, and a digit at that location.

**Solution 17.4:** 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(A, 0, 0)) {
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(vector<vector<int>>& A, int i, int j) {
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(A, i + 1, j);
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(A, i + 1, j)) {
41                 return true;
42             }
43         }
44     }
45
46     (*A)[i][j] = 0; // undos 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 }

```

**Variant 17.4.1:** Compute a placement of eight queens on an  $8 \times 8$  chessboard in which no two queens attack each other.

**Variant 17.4.2:** Compute a placement of 32 knights, or 14 bishops, 16 kings or eight rooks on an  $8 \times 8$  chessboard in which no two pieces attack each other.

**Variant 17.4.3:** Compute the smallest number of queens that can be placed to attack each uncovered square.

**Problem 18.1, pg. 96:** Design an online spell correction system. It should take as input a string  $s$  and return an array of entries in its dictionary which are closest to the string using the Levenshtein distance specified in Problem 15.3 on Page 82. Cache the most recently computed result.

**Solution 18.1:** The naïve solution would be:

```

1 public class S1 extends SpellCheckService {
2     static String wLast = null;
3     static String [] closestToLastWord = null;
4
5     public static void service(ServiceRequest req, ServiceResponse resp) {
6         String w = req.extractWordToCheckFromRequest();
7         if (!w.equals(wLast)) {
8             wLast = w;
9             closestToLastWord = Spell.closestInDictionary(w);
10        }
11        resp.encodeIntoResponse(closestToLastWord);
12    }
13 }

```

This solution has a race condition. Suppose Threads  $A$  and  $B$  run the service. Suppose Thread  $A$  updates  $wLast$ , and then Thread  $B$  is scheduled. Now Thread  $B$  reads  $wLast$  and  $closestToLastWord$ . Since Thread  $A$  has not updated  $closestToLastWord$ , if  $wLast$  equals the check string  $w$  passed to  $B$ , the cached  $closestToLastWord$   $B$  returns corresponds to the previous value of  $wLast$ . The call to  $closestToLastWord$  could take quite long or be very fast, depending on the length and contents of  $checkWord$ . Hence it is entirely possible that Thread  $B$  reads both  $wLast$  and  $closestToLastWord$  between Thread  $A$ 's updates them.

A thread-safe solution would be to declare `service` to be synchronized; in this case, only one thread could be executing the method and there is no race between

write to `wLast` and `closestToLastWord`. This leads to poor performance—only one thread can be executing at a time.

The solution is to lock just the part of the code that operates on the cached values—specifically, the check on the cached value and the updates to the cached values:

```

1 public class S2 extends SpellCheckService {
2     static String wLast = null;
3     static String [] closestToLastWord = null;
4
5     public static void service(ServiceRequest req, ServiceResponse resp) {
6         String w = req.extractWordToCheckFromRequest();
7         String [] result = null;
8         synchronized (S2.class) {
9             if (w.equals(wLast)) {
10                 result = Arrays.copyOf(closestToLastWord, closestToLastWord.length);
11             }
12         }
13         if (result == null) {
14             result = Spell.closestInDictionary(w);
15             synchronized (S2.class) {
16                 wLast = w;
17                 closestToLastWord = result;
18             }
19         }
20         resp.encodeIntoResponse(result);
21     }
22 }

```

In the above code, multiple threads can be in their call to `closestInDictionary` which is good because the call may take a long time. Locking ensures that the read assignment on a hit and write assignment on completion are atomic. Note that we have to clone `closestToLastWord` when assigning to `result` since otherwise, `closestToLastWord` might change before we encode it into the response.

**Variant 18.1.1:** Threads 1 to  $n$  execute a method called `critical`. Before this, they execute a method called `rendezvous`. The synchronization constraint is that only one thread can execute `critical` at a time, and all threads must have completed executing `rendezvous` before `critical` can be called. You can assume  $n$  is stored in a variable `n` that is accessible from all threads. Design a synchronization mechanism for the threads. All threads must execute the same code. Threads may call `critical` multiple times, and you should ensure that a thread cannot call `critical` a  $(k + 1)$ -th time until all other threads have completed their  $k$ -th calls to `critical`.

**Variant 18.1.2:** In this problem you are to design a synchronization mechanism for a pool. This is a data structure that combines requests. Specifically, requests come from two types of threads. The pool has a capacity of four requests. A thread cannot have more than one request in the pool. When the pool is full, it must be the case that requests from both types of threads are present. Exactly one of the requesting threads must call the `launch` function when four requests are in the pool. Each thread

corresponding to a request in the pool should invoke a flush function before launch is executed. Threads should call flush as late as possible.

**Problem 18.2, pg. 96:** *Develop a `Timer` class that manages the execution of deferred tasks. The `Timer` constructor takes as its argument an object which includes a `Run` method and a `name` field, which is a string. `Timer` must support—(1.) starting a thread, identified by name, at a given time in the future; and (2.) canceling a thread, identified by name (the cancel request is to be ignored if the thread has already started).*

**Solution 18.2:** 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 18.3, pg. 96:** *Implement a synchronization mechanism for the first readers-writers problem.*

**Solution 18.3:** We want to indicate whether the string is being read as well as whether the string is being written to. We achieve this with a pair of locks—LR and LW and a read counter locked by LR.

A reader proceeds as follows. It locks LR, increments the counter, and releases LR. After it performs its reads, it locks LR, decrements the counter, and releases LR. A writer locks LW, then performs the following in an infinite loop. It locks LR, checks to see if the read counter is 0; if so, it performs its write, releases LR, and breaks out of the loop. Finally, it releases LW. In the code below we use the Java `wait()` and `notify()` primitives to avoid the CPU cycles wasted in a busy wait.

```
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 19.1, pg. 98:** *Design a program that produces high quality mosaics with minimal compute time.*

**Solution 19.1:** A good way to begin is to partition the image into  $s \times s$ -sized squares, compute the average color of each such image square, and then find the tile that is closest to it in the color space. Distance in the color space can be the  $L_2$ -distance over the Red-Green-Blue (RGB) intensities for the color. As you look more carefully at the problem, you might conclude that it would be better to match each tile with



an image square that has a similar structure. One way could be to perform a coarse pixelization ( $2 \times 2$  or  $3 \times 3$ ) of each image square and finding the tile that is “closest” to the image square under a distance function defined over all pixel colors. In essence, the problem reduces to finding the closest point from a set of points in a  $k$ -dimensional space.

Given  $m$  tiles and an image partitioned into  $n$  squares, then a brute-force approach would have  $O(mn)$  time complexity. You could improve on this by first indexing the tiles using an appropriate search tree. You can also run the matching in parallel by partitioning the original image into subimages and searching for matches on the subimages independently.

**Problem 19.2, pg. 98:** *Design a system that can compute the ranks of ten billion web pages in a reasonable amount of time.*

**Solution 19.2:** 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  $X_i$  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 20.1, pg. 101:** Let  $A$  be an array of  $n$  distinct elements. Design an algorithm that returns a subset of  $k$  elements of  $A$ . All subsets should be equally likely. Use as few calls to the random number generator as possible and use  $O(1)$  additional storage. You can return the result in the same array as input.

**Solution 20.1:** The problem is trivial when  $k = 1$ —we simply make one call to the random number generator, take the returned  $r$  value mod  $n$ . We can swap  $A[n - 1]$  with  $A[r]$ ;  $A[n - 1]$  then holds the result.

For  $k > 1$ , we start by choosing one element at random as above and we now repeat the same process with the  $n - 1$  element subarray  $A[0 : n - 2]$ . Eventually, the random subset occupies the slots  $A[n - k : n - 1]$  and the remaining elements are in the first  $n - k$  slots.

The algorithm clearly runs in  $O(1)$  space. To show that all the subsets are equally likely, we prove something stronger, namely that all permutations of size  $k$  are equally likely.

Formally, an  $m$ -permutation of a set  $S$  of cardinality  $n$  is a sequence of  $m$  elements of  $S$  with no repetitions. It is easily verified that the number of  $m$ -permutations is  $\frac{n!}{(n-m)!}$ .

The induction hypothesis now is that after iteration  $m$ , the subarray  $A[n - m : n - 1]$  contains each possible  $m$ -permutation with probability  $\frac{(n-m)!}{n!}$ .

The base case holds since for  $m = 1$ , any element is equally likely to be selected.

Suppose the inductive hypothesis holds for  $m = l$ . Now we study  $m = l + 1$ . Consider a particular  $(l + 1)$ -permutation, say  $\langle \alpha_1, \dots, \alpha_{l+1} \rangle$ . This consists of a single element  $\alpha_1$  followed by the  $l$ -permutation  $\langle \alpha_2, \dots, \alpha_{l+1} \rangle$ . Let  $E_1$  be the event that  $\alpha_1$  is selected in iteration  $l + 1$  and  $E_2$  be the event that the first  $l$  iterations produced  $\langle \alpha_2, \dots, \alpha_{l+1} \rangle$ . The probability of  $\langle \alpha_1, \dots, \alpha_{l+1} \rangle$  resulting after iteration  $l + 1$  is simply  $\Pr(E_1 \cap E_2) = \Pr(E_1 | E_2)\Pr(E_2)$ . By the inductive hypothesis, the probability of permutation  $\langle \alpha_2, \dots, \alpha_{l+1} \rangle$  is  $\frac{(n-l)!}{n!}$ . The probability  $\Pr(E_1|E_2) = \frac{1}{n-l}$  since the algorithm selects from elements in the subarray  $A[0 : n - l - 1]$  with equal probability. Therefore

$$\Pr(E_1 \cap E_2) = \Pr(E_1|E_2)\Pr(E_2) = \frac{1}{n-l} \frac{(n-l)!}{n!} = \frac{(n-l-1)!}{n!}$$

and induction goes through.

The algorithm generates all random  $k$ -permutations with equal probability, from which it follows that all subsets of size  $k$  are equally likely.

The algorithm just described makes  $k$  calls to the random number generator. When  $k$  is bigger than  $\frac{n}{2}$ , we can optimize by computing a subset of  $n - k$  elements to remove from the set. For example, when  $k = n - 1$ , this replaces  $n - 1$  calls to the random number generator with a single call. Of course, while all subsets are equally likely with this optimization, all permutations are not. Following is the code in C++:

```
1 vector<int> offline_sampling(vector<int> A, int k) {
```

```

2  for (int i = 0; i < k; ++i) {
3      default_random_engine gen((random_device())()); // random num generator.
4      // Generate 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 20.1.1:** 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 20.2, pg. 101:** How would you implement a random number generator that generates a random integer  $i$  in  $[a, b]$ , given a random number generator that produces either zero or one with equal probability? All generated values should have equal probability. What is the run time of your algorithm, assuming each call to the given random number generator takes  $O(1)$  time?

**Solution 20.2:** Basically, we want to produce a random integer in  $[0, b - a]$ . Let  $l = b - a + 1$ . We can produce a random integer in  $[0, l - 1]$ , as follows. Let  $i$  be the least integer such that  $l \leq 2^i$ .

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

If  $l$  is not a power of 2, the  $i$  calls may or may not encode an integer in the range 0 to  $l - 1$ . If the number is in the range, we return it; since all the numbers are equally likely, the result is correct.

If the number is outside the range  $[0, l - 1]$ , we try again. The probability of having to try again is less than  $\frac{1}{2}$  since  $l > 2^{i-1}$ . Therefore the probability that we take exactly  $k$  steps before succeeding is at most  $\frac{1}{2}(1 - \frac{1}{2})^{k-1} = \frac{1}{2}^k$ . This implies the expected number of trials is less than  $1\frac{1}{2} + 2(\frac{1}{2})^2 + 3(\frac{1}{2})^3 + \dots$ . Differentiating the identity  $\frac{1}{1-x} = 1 + x + x^2 + x^3 + \dots$ , yields the identity  $\frac{1}{(1-x)^2} = 1 + 2x + 3x^2 + 4x^3 + \dots$ . Multiplying both sides by  $x$  demonstrate that  $\frac{x}{(1-x)^2} = x + 2x^2 + 3x^3 + 4x^4 + \dots$ . Substituting  $\frac{1}{2}$  for  $x$  in this last identity proves that  $1(\frac{1}{2}) + 2(\frac{1}{2})^2 + 3(\frac{1}{2})^3 + \dots = \frac{\frac{1}{2}}{(1-\frac{1}{2})^2} = 2$ . Therefore the expected number of trials is less than 2.

```

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

```

```

10 |     return res + a;
11 | }

```

**Problem 20.3, pg. 101:** Design an algorithm that reads a sequence of packets and maintains a uniform random subset of size  $k$  of the read packets when the  $n \geq k$ -th packet is read.

**Solution 20.3:** We store the first  $k$  packets. Consequently, we select the  $n$ -th packet to add to our subset with probability  $\frac{k}{n}$ . If we do choose it, we select an element uniformly at random to eject from the subset.

To prove correctness, we use induction on the number of packets that have been read. Specifically, the inductive hypothesis is that all  $k$ -sized subsets are equally likely after  $n \geq k$  packets have been read.

The number of  $k$ -size subsets is  $\binom{n}{k}$ , implying the probability of any  $k$ -size subset should be  $\frac{1}{\binom{n}{k}}$ .

For the base case,  $n = k$ , there is exactly one subset of size  $k$  which is what the algorithm computes.

Assume the induction hypothesis holds for  $n > k$ . Consider the  $(n + 1)$ -th packet. The probability of a  $k$ -size subset that does not include the  $(n + 1)$ -th packet is the probability that the  $k$ -size subset was selected after reading the  $n$ -th packet and the  $(n + 1)$ -th packet was not selected. These two events are independent, which means the probability of selecting such a subset is

$$\frac{1}{\binom{n}{k}} \left(1 - \frac{k}{n+1}\right) = \frac{k!(n-k)!}{n!} \left(\frac{n+1-k}{n+1}\right) = \frac{k!(n+1-k)!}{(n+1)!}.$$

This simplifies to  $\frac{1}{\binom{n+1}{k}}$ , so induction goes for subsets excluding the  $n + 1$  element.

The probability of a  $k$ -size subset  $H$  that includes the  $(n + 1)$ -th packet  $p_{n+1}$  can be computed as follows. Let  $G$  be a  $k$ -size subset of the first  $n$  packets. The only way we can get from  $G$  to  $H$  is if  $G$  contains  $H \setminus \{p_{n+1}\}$ . Let  $G^*$  be such a subset; let  $\{q\} = G \setminus G^*$ .

The probability of going from  $G$  to  $H$  is the probability of selecting  $p_{n+1}$  and dropping  $q$ , which is equal to  $\frac{k}{n+1} \cdot \frac{1}{k}$ . There exist  $-(k - 1)$  candidate subsets for  $G^*$ , each with probability  $\frac{1}{\binom{n}{k}}$  (by the inductive hypothesis) which means that the probability of  $H$  is given by

$$\frac{k}{n+1} \frac{1}{k} (n + (k - 1)) \frac{1}{\binom{n}{k}} = \frac{(n+1-k)(n-k)!k!}{(n+1)n!} = \frac{1}{\binom{n+1}{k}},$$

so induction goes through for subsets including the  $(n + 1)$ -th element. Following is the code in C++:

```

1 | vector<int> reservoir_sampling(istream& 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 + 1;
11  while (*sin >> x) {
12      default_random_engine gen((random_device())()); // random num generator.
13      // Generate 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 }

```

**Problem 20.4, pg. 101:** Design an algorithm that computes an array of size  $k$  consisting of distinct integers in the set  $\{0, 1, \dots, n - 1\}$ . All subsets should be equally likely and, in addition, all permutations of elements of the array should be equally likely. Your time should be  $O(k)$ . Your algorithm should use  $O(k)$  space in addition to the  $k$  element array holding the result. You may assume the existence of a subroutine that returns integers in the set  $\{0, 1, \dots, n - 1\}$  with uniform probability.

**Solution 20.4:** We mimic the offline sampling algorithm described in Solution 20.1 on Page 190, with  $A[i] = i$  initially. Since the array  $A$  is of length  $n$ , and most of it is unchanged, when  $k \ll n$ , we simulate  $A$  with a hash table.

Specifically, we maintain a hash table  $H$ —both keys and values are from the set  $\{0, 1, \dots, n - 1\}$ . Conceptually,  $H$  tracks indices of the array for which  $A[i]$  may not equal  $i$ . Initially  $H$  is empty. Denote the key associated with the value  $v$  by  $H(v)$ .

We do  $k$  iterations of the following. Choose a random integer  $r$  in  $[0, n - 1 - i]$ , where  $i$  is the current iteration count, starting at 0. There are four possibilities.

- $r$  is not a key in  $H$  and  $n - 1 - i$  is not a key in  $H$ : add the key-value pairs  $(n - 1 - i, r)$  and  $(r, n - 1 - i)$  to  $H$
- $r$  is not a key in  $H$  and  $n - 1 - i$  is a key in  $H$ : add the key-value pairs  $(r, H(n - 1 - i))$  and  $(n - 1 - i, r)$  to  $H$  (this will overwrite the value associated with  $n - 1 - i$ ).
- $r$  is a key in  $H$  and  $n - 1 - i$  is not a key in  $H$ : add the key-value pairs  $(n - 1 - i, H(r))$  and  $(r, (n - 1 - i))$  to  $H$ .
- $r$  is a key in  $H$  and  $n - 1 - i$  is a key in  $H$ : add the key-value pairs  $(r, H(n - 1 - i))$  and  $(n - 1 - i, H(r))$  to  $H$ .

The desired result is in  $A[n - k : n - 1]$ , which can be determined from  $H$ .

```

1  vector<int> online_sampling(int n, int k) {
2      unordered_map<int, int> H;
3      default_random_engine gen((random_device())()); // random num generator.
4      for (int i = 0; i < k; ++i) {
5          // Generate random int in [i, n - 1].
6          uniform_int_distribution<int> dis(0, n - 1 - i);
7          int r = dis(gen);
8          auto ptr1 = H.find(r), ptr2 = H.find(n - 1 - i);

```

```

9   if (ptr1 == H.end() && ptr2 == H.end()) {
10       H[r] = n - 1 - i;
11       H[n - 1 - i] = r;
12   } else if (ptr1 == H.end() && ptr2 != H.end()) {
13       H[r] = ptr2->second;
14       ptr2->second = r;
15   } else if (ptr1 != H.end() && ptr2 == H.end()) {
16       H[n - 1 - i] = ptr1->second;
17       ptr1->second = n - 1 - i;
18   } else {
19       int temp = ptr2->second;
20       H[n - 1 - i] = ptr1->second;
21       H[r] = temp;
22   }
23 }
24 vector<int> res;
25 for (int i = 0; i < k; ++i) {
26     res.emplace_back(H[n - 1 - i]);
27 }
28 return res;
29 }

```

**Problem 21.1, pg. 103:** Which of the 500 doors are open after the 500-th person has walked through?

**Solution 21.1:** As described on Page 37, analyzing a few small examples suggests that, independent of  $n$ , door  $k$  will be open iff  $k$  is a perfect square. This can be rigorously proved as follows.

**Proof:**

If the number of times a door's state changes is odd, it will be open; otherwise it is closed. Therefore the number of times door  $k$ 's state changes equals the number of divisors of  $k$ . From the small example analysis, we are led to the conjecture that the number of divisors of  $k$  is odd iff  $k$  is a perfect square. Note that if  $d$  divides  $k$ , then  $k/d$  also divides  $k$ . Therefore we can uniquely pair off divisors of  $k$ , other than  $\sqrt{k}$  (if it is an integer). Hence, when  $\sqrt{k}$  is not an integer,  $k$  has an even number of divisors. When  $\sqrt{k}$  is an integer, it is the only divisor of  $k$  that cannot be uniquely paired off with another divisor, implying  $k$  has an odd number of divisors. By definition,  $\sqrt{k}$  is an integer iff  $k$  is a perfect square, proving the result.

This check can be performed by squaring  $\lfloor \sqrt{i} \rfloor$  and comparing the result with  $i$ .

```

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 21.1.1:** There are 25 people seated at a round table. Each person has two cards. Each card has a number from 1 to 25. Each number appears on exactly two

cards. Each person passes the card with the smaller number to the person on his left. This is done iteratively in a synchronized fashion. Show that eventually someone will have two cards with identical numbers.

**Problem 21.2, pg. 103:** *What is the minimum number of five man time-trials needed to determine the top three cyclists from a set of 25 cyclists?*

**Solution 21.2:** Let's start with five time-trials with no cyclist being in more than one of these five initial time-trials. Let the rankings be  $\langle A1, A2, A3, A4, A5 \rangle$ ,  $\langle B1, B2, B3, B4, B5 \rangle$ ,  $\langle C1, C2, C3, C4, C5 \rangle$ ,  $\langle D1, D2, D3, D4, D5 \rangle$ , and  $\langle E1, E2, E3, E4, E5 \rangle$ , where the first cyclist in each sequence is the fastest. Note that we can eliminate  $A4, A5, B4, B5, C4, C5, D4, D5, E4$ , and  $E5$  at this stage.

Now, we race the winners from each of the initial time-trials. Without loss of generality, assume the outcome is  $\langle A1, B1, C1, D1, E1 \rangle$ . At this point, we can eliminate  $D1$  and  $E1$  as well as  $D2$  and  $D3$  and  $E2$  and  $E3$ . Furthermore, since  $C1$  was third,  $C2$  and  $C3$  cannot be in the top three; Similarly,  $B3$  cannot be a contender.

We need to find the best and the second best from  $A2, A3, B1, B2$ , and  $C1$ , which we can determine with one more time-trial. Therefore seven time-trials are enough.

Note that we need six time-trials to determine the overall winner, and the sequence of time-trials to determine the winner is essentially unique—if some cyclists did not participate in the first five time-trials, he would have to participate in the sixth one. But then one of the winners of the first five time-trials would not participate in the sixth time-trial and he might be the overall winner. The first six time-trials do not determine the second and the third fastest cyclists, hence a seventh race is necessary.

**Problem 21.3, pg. 104:** *Given an instance of the gasup problem, how would you efficiently compute an ample city if one exists?*

**Solution 21.3:** Consider the thought experiment of starting at an arbitrary city with sufficiently large amount of gas so that we can complete the loop. In this experiment, we note the amount of gas in the tank as the vehicle goes through the loop at each city before loading the gas kept in that city for the vehicle. Let  $C$  be a city where the amount of gas in the tank before we refuel at that city is minimum. Call this minimum amount of gas  $m$ . Now suppose we pick  $C$  as the starting point, and we have no gas. Since we never have less gas than we started with at  $C$ , we can complete the journey without running out of gas. The computation to determine  $C$  can be easily done in linear time with a single pass over all the cities.

```

1 int find_start_city(const vector<int>& G, const vector<int>& D) {
2     int carry = 0;
3     pair<int, int> min(0, 0);
4     for (int i = 1; i < G.size(); ++i) {
5         carry += G[i - 1] - D[i - 1];
6         if (carry < min.second) {
7             min = {i, carry};
8         }
9     }
10    return min.first;

```

11 }

**Problem 21.4, pg. 104:** Given an array  $A$  with  $n$  elements, compute  $\max_{j=0}^{n-1} \frac{\prod_{i=0}^{n-1} A[i]}{A[j]}$  in  $O(n)$  time without using division. Can you design an algorithm that runs in  $O(1)$  space and  $O(n)$  time? Array entries may be positive, negative, or 0.

**Solution 21.4:** Let  $L_p = \prod_{i=0}^p A[i]$  and  $R_p = \prod_{j=p}^{n-1} A[j]$ . Observe computing  $L_p$  and  $R_p$  individually takes  $p$  and  $(n-1) - p$  multiplications, respectively; however, we can compute all the  $L_p$  and  $R_p$  using  $2(n-1)$  multiplications, since  $L_p = L_{p-1}A[p]$ , and  $R_p = A[p]R_{p+1}$ . The product of all elements except the  $i$ -th one is simply  $L_{i-1}R_{i+1}$ , hence we can compute these  $n$  products using  $n$  multiplications, once we have  $L$  and  $R$  computed. Finding the largest product is simply an iteration with compare and swap in the loop. The time complexity is  $O(n)$  and the solution uses two arrays of length  $n$  each.

```

1 int find_biggest_n_1_product(const vector<int>& A) {
2     // Build forward product L, and backward product R.
3     vector<int> L, R(A.size());
4     partial_sum(A.cbegin(), A.cend(), back_inserter(L), multiplies<int>());
5     partial_sum(A.crbegin(), A.crend(), R.rbegin(), multiplies<int>());
6
7     // Find the biggest product of (n - 1) numbers.
8     int max_product = numeric_limits<int>::min();
9     for (int i = 0; i < A.size(); ++i) {
10         int forward = i > 0 ? L[i - 1] : 1;
11         int backward = i + 1 < A.size() ? R[i + 1] : 1;
12         max_product = max(max_product, forward * backward);
13     }
14     return max_product;
15 }

```

It is possible to solve this problem with only  $O(1)$  additional storage, with a sophisticated case analysis. Suppose  $A[i] \neq 0$  for all  $i$ . If  $A$  contains an odd number of negative entries, the optimum product is formed when we exclude the biggest negative entry. Otherwise, suppose  $A$  contains an even number of negative entries. If some entries are positive, the optimum product is achieved when we exclude the smallest positive entry; otherwise, we exclude the smallest negative number.

Now suppose two or more zeros are present in  $A$ . Then the product of any  $n-1$  entries is always 0. Suppose there is exactly one zero. If  $A$  contains an odd number of negative numbers, the optimum product is zero; otherwise it is the product of all elements excluding the zero. The code can be readily implemented with a small number of traversals of the array, leading to an  $O(n)$  time complexity.

```

1 int find_biggest_n_1_product(const vector<int>& A) {
2     int zero_count = 0, pos_count = 0, neg_count = 0;
3     int zero_idx = -1, s_neg_idx = -1, b_neg_idx = -1, s_pos_idx = -1;
4
5     for (int i = 0; i < A.size(); ++i) {
6         if (A[i] < 0) {

```



```
7         ++neg_count;
8         if (s_neg_idx == -1 || A[i] < A[s_neg_idx]) {
9             s_neg_idx = i;
10        }
11        if (b_neg_idx == -1 || A[b_neg_idx] < A[i]) {
12            b_neg_idx = i;
13        }
14    } else if (A[i] == 0) {
15        zero_idx = i, ++zero_count;
16    } else { // A[i] > 0.
17        ++pos_count;
18        if (s_pos_idx == -1 || A[i] < A[s_pos_idx]) {
19            s_pos_idx = i;
20        }
21    }
22 }
23
24 // Try to find a number whose elimination could maximize the product of
25 // the remaining (n - 1) numbers.
26 int x; // stores the idx of eliminated one.
27 if (zero_count >= 2) {
28     return 0;
29 } else if (zero_count == 1) {
30     if (neg_count & 1) {
31         return 0;
32     } else {
33         x = zero_idx;
34     }
35 } else {
36     if (neg_count & 1) { // odd number negative.
37         x = b_neg_idx;
38     } else { // even number negative.
39         if (pos_count > 0) {
40             x = s_pos_idx;
41         } else {
42             x = s_neg_idx;
43         }
44     }
45 }
46
47 int product = 1;
48 for (int i = 0; i < A.size(); ++i) {
49     if (i != x) {
50         product *= A[i];
51     }
52 }
53 return product;
54 }
```

**Variant 21.4.1:** Let  $A$  be as above. Compute an array  $B$  where  $B[i]$  is the product of all elements in  $A$  except  $A[i]$ . You cannot use division. Your time complexity should be  $O(n)$ , and you can only use  $O(1)$  additional space.

**Problem 21.5, pg. 105:** Given  $c$  cases and  $d$  drops, what is the maximum number of floors that you can test in the worst-case?

**Solution 21.5:** Let  $F(c, d)$  be the maximum number of floors we can test with  $c$  identical cases and at most  $d$  drops. We know that  $F(1, d) = d$ . Suppose we know the value of  $F(i, j)$  for all  $i \leq c$  and  $j \leq d$ .

If we are given  $c + 1$  cases and  $d$  drops we can start at floor  $F(c, d - 1) + 1$  and drop a case. If the case breaks, then we can use the remaining  $c$  cases and  $d - 1$  drops to determine the floor exactly, since it must be in the range  $[1, F(c, d - 1)]$ . If the case did not break, we proceed to floor  $F(c, d - 1) + 1 + F(c + 1, d - 1)$ .

Therefore  $F$  satisfies the recurrence

$$F(c + 1, d) = F(c, d - 1) + 1 + F(c + 1, d - 1).$$

We can compute  $F$  using DP as below:

```

1 int getHeight(int c, int d) {
2     vector<vector<int>> F(c + 1, vector<int>(d + 1, -1));
3     return get_height_helper(&F, c, d);
4 }
5
6 int get_height_helper(vector<vector<int>>*& F, int c, int d) {
7     if (d == 0) {
8         return 0;
9     } else if (c == 1) {
10        return d;
11    } else {
12        if ((*F)[c][d] == -1) {
13            (*F)[c][d] = get_height_helper(F, c, d - 1) +
14                        get_height_helper(F, c - 1, d - 1) + 1;
15        }
16        return (*F)[c][d];
17    }
18 }

```

**Variant 21.5.1:** How would you compute the minimum number of drops needed to find the breaking point from 1 to  $F$  floors using  $c$  cases?

**Variant 21.5.2:** Men numbered from 1 to  $n$  are arranged in a circle in clockwise order. Every  $k$ -th man is removed, until only one man remains. What is the number of the last man?

**Problem 21.6, pg. 105:** Design an algorithm for pairing bidders with celebrities to maximize the revenue from the dance. Each celebrity cannot dance more than once, and each bidder cannot dance more than once. Assume that the set of celebrities is disjoint from the set of bidders. How would you modify your approach if all bids were for the same amount? What if celebrities and bidders are not disjoint?

**Solution 21.6:** The problem can directly be mapped into the weighted bipartite matching problem. Bidders and celebrities constitute the left and right vertices; an edge exists from  $b$  to  $c$  iff  $b$  has offered money to dance with  $c$ , and the weight of an edge is the amount offered for the dance. It can be solved using specialized algorithms, flow network, or linear programming.

If the bids are all in the same amount, the problem is that of unweighted bipartite matching. If the requirement that bidders and celebrities be distinct is dropped, the problem becomes a weighted matching problem in a general graph. Both of these variants are solvable in polynomial time.

## Part V

### Notation and Index



# Notation

*To speak about notation as the only way that you can guarantee structure of course is already very suspect.*

— E. S. PARKER

We use the following convention for symbols, unless the surrounding text specifies otherwise:

$i, j, k$	nonnegative array indices
$f, g, h$	function
$A$	$k$ -dimensional array
$L$	linked list or doubly linked list
$S$	set
$T$	tree
$G$	graph
$V$	set of vertices of a graph
$E$	set of edges of a graph
$\mathcal{E}$	an event from a probability space
$u, v$	vertex-valued variables
$e$	edge-valued variable
$m, n$	number of elements in a collection
$x, y$	real-valued variables
$\sigma$	a permutation

Symbolism	Meaning
$(d_{k-1} \dots d_0)_r$	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 $x$
$\langle a_0, a_1, \dots, a_{n-1} \rangle$	sequence of $n$ elements
$a^k, a = \langle a_0, \dots, a_{n-1} \rangle$	the sequence $\langle a_k, a_{k+1}, \dots, a_{n-1} \rangle$
$\sum_{R(k)} f(k)$	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 $\sum_{a \leq k \leq b} f(k)$
$\prod_{k=a}^b f(k)$	shorthand for $\prod_{a \leq k \leq b} f(k)$
$\{a \mid R(a)\}$	set of all $a$ such that the relation $R(a) = \text{true}$
$[l, r]$	closed interval: $\{x \mid l \leq x \leq r\}$
$[l, r)$	half-closed, half-open interval: $\{x \mid l \leq x < r\}$
$(l, r]$	half-open, half-closed interval: $\{x \mid l < x \leq r\}$
$(l, r)$	open interval: $\{x \mid l < x < r\}$
$\{a, b, \dots\}$	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 two-dimensional array $A$
$A[i_1 : i_2][j_1 : j_2]$	2D subarray of two-dimensional array $A$ consisting of elements from $i_1$ -th to $i_2$ -th rows and from $j_1$ -th to $j_2$ -th column, inclusive
$\binom{n}{k}$	binomial coefficient: number of ways of choosing $k$ elements from a set of $n$ items
$n!$	$n$ -factorial, the product of the integers from 1 to $n$ , inclusive
$O(f(n))$	big-oh complexity of $f(n)$ , asymptotic upper bound
$\Theta(f(n))$	big-theta complexity of $f(n)$ , asymptotically tight bound
$\Omega(f(n))$	big-Omega complexity of $f(n)$ , asymptotic lower bound
$x \bmod y$	mod function
$x \oplus y$	bitwise-XOR function
$x \approx y$	$x$ is approximately equal to $y$
null	pointer value reserved for indicating that the pointer does not refer to a valid address
$\emptyset$	empty set
$\infty$	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.
$\mathbb{Z}$	the set of integers $\{\dots, -2, -1, 0, 1, 2, 3, \dots\}$
$\mathbb{Z}^+$	the set of nonnegative integers $\{0, 1, 2, 3, \dots\}$
$\mathbb{Z}_n$	the set $\{0, 1, 2, 3, \dots, n-1\}$
$\mathbb{R}$	the set of real numbers
$\mathbb{R}^+$	the set of nonnegative real numbers
$x \ll y$	much less than
$x \gg y$	much greater than
$A \mapsto B$	function mapping from domain $A$ to range $B$
$\Rightarrow$	logical implication
iff	if and only if
$\Pr(\mathcal{E})$	probability of event $\mathcal{E}$



---

## Index of Terms

- 2D array, 36, 83, 88, 170, 171, 175, 183
- $O(1)$  space, 24, 25, 43, 50, 51, 61, 73, 81, 117, 120, 125, 142, 160, 190
- 0-1 knapsack problem, 93, 181
- abstract analysis patterns, 22, 36
- abstract data type, *see* ADT
- adjacency list, 87, 87, 189
- adjacency matrix, 87, 87
- ADT, 24, 24, 25, 26, 56, 57
- AKS primality testing, 43
- algorithm design patterns, 22, 27
- all pairs shortest paths, 88
- alternating sequence, 165
- amortized, 56
- amortized analysis, 49, 71
- API, 25, 25, 58, 132
- application programming interface, *see* API
- approximation, 28, 34, 35, 78, 98, 100
- approximation algorithm, 91
- arbitrage, 40, 40
- array, 1–3, 11, 13, 23, 23, 24, 25, 29–31, 33–35, 39, 43, 49, 49, 50, 56–58, 64, 66, 68, 69, 71, 73–77, 80–82, 96, 101, 102, 104, 105, 112, 117, 119, 132, 136, 137, 141–145, 147, 150, 152, 154, 155, 158, 162, 164–166, 181, 182, 185, 190, 193, 196, 197
  - 2D, *see* 2D array
  - bit, *see* bit array
  - deletion from, 49
- ascending sequence, 166
- auxiliary elements, 42
- AVL tree, 26, 27
- backtracking, 78, 183
- balanced BST, 27, 63
- Bellman-Ford algorithm, 40
- Bernoulli random variable, 100
- BFS, 87, 87, 161, 163, 176, 177
- BFS tree, 176
- binary search, 3, 13, 20, 32, 33, 66, 66, 67, 68, 73, 79, 91, 143, 145, 146, 154, 165
- binary search tree, 3, 23, 23, 25, *see* BST, 71
  - AVL tree, 26, 27
  - deletion from, 23, 26
  - height of, 76, 77, 162
  - red-black tree, 26, 27, 76
- binary tree, *see also* binary search tree, 23, 25, 26, 57, 59–63, 76, 87, 131, 133, 135, 159, 172, 174
  - complete, 60, 63
  - full, 60
  - height of, 23, 25, 26, 60, 61
  - perfect, 60
- binomial coefficient, 40
- bipartite graph, 89
- bipartite matching, 199
- bit array, 20, 113, 147
- bitonic sequence, 166, 166
- Bloom filter, 23
- Boost, 12
- Boyer-Moore algorithm, 39
- branch and bound, 91
- breadth-first search, *see* BFS
- brute-force solution, 37
- BST, 12, 26, 26, 27, 30, 50, 76, 77, 159–162
- busy wait, 187
- caching, 28, 33
- capacity constraint, 93
- cardinality, 190
- case analysis, 36, 36
- central processing unit, *see* CPU
- chessboard, 30, 41, 42, 185
  - mutilated, 29
- child, 57, 60, 76, 87, 134, 172, 174, 175
- circular queue, *see also* queue
- closed interval, 27, 156
- CNF-SAT, 91, 91
- code
  - hash, *see* hash code

- Huffman, 84, 172–175
- coin changing, 83
- coloring, 41
- combination, 31, 82, 83, 169
- complete binary tree, 60, 60, 63
  - height of, 60
- complex number, 46
- complexity analysis, 43
- concurrency, 4
- conjunctive normal form satisfiability, *see* CNF-SAT
- connected component, 27, 86, 86
- connected directed graph, 86
- connected undirected graph, 86, 86, 89
- connected vertices, 86, 86
- constraint, 1, 25, 32, 41, 89, 93, 97, 105, 159, 182
  - capacity, 93
  - space, 35
  - synchronization, 186
- convex sequence, 166
- counting sort, 50, 50
- CPU, 33, 37
- cumulative distribution function, 100
- DAG, 85, 178
- data structure, 22
- data structure patterns, 22, 22
- database, 33
- deadlock, 96
- decrease and conquer, 79, 144
- degree
  - of a node in a rooted tree, 163
  - of a polynomial, 43, 158
  - of a subtree, 163
- deletion
  - from arrays, 49
  - from binary search trees, 23, 26
  - from doubly linked lists, 57
  - from hash tables, 23, 71
  - from heaps, 23
  - from linked list, 23
  - from max-heaps, 63
  - from priority queues, 26
  - from queues, 23
  - from stacks, 23
- depth
  - of a node in a binary search tree, 161
  - of a node in a binary tree, 23, 60, 60
  - of a node in a Huffman tree, 174
  - of the function call stack, 44, 114
- depth-first search, 44, *see* DFS
- deque, 57
- dequeue, 57
- DFS, 87, 87, 175, 176, 179
- diameter
  - of a tree, 79, 80, 163
- Dijkstra's algorithm, 3, 27, 179
  - implemented with a Fibonacci heap, 27
- directed acyclic graph, *see* DAG, 86
- directed graph, 85, *see also* directed acyclic graph, *see also* graph, 85, 86, 89, 179
  - connected directed graph, 86
  - weakly connected graph, 86
- discovery time, 87
- disjoint-set data structure, 27, 27
- distance
  - Levenshtein, 36, 82, 96, 166–168, 185
- distributed memory, 95, 96
- distribution
  - of the elements, 35
  - of the inputs, 44
  - of the numbers, 33
- divide and conquer, 2, 3, 12, 28, 29, 30, 39, 78–81, 163
- divisor, 194
  - greatest common divisor, 47
- double-ended queue, *see* deque
- doubly linked list, 23, *see also* linked list, 24, 53, 53, 57, 123, 201
  - deletion from, 57
- DP, 12, 30, 30, 31, 33, 41, 80, 81, 84, 91, 168–171, 181, 198
- dynamic order statistics, 23
- dynamic programming, 3, *see* DP, 30, 80
- edge, 40, 41, 79, 85, 85, 87, 89, 90, 174, 178, 179, 199, 201
- elimination, 28, 32, 66
- enqueue, 57
- Ethernet, 79
- expected value, 100, 100
- extract-max, 63
- extract-min, 136, 139, 140, 172
- fast Fourier Transform, *see* FFT
- FFT, 158
- Fibonacci heap, 27
  - in Dijkstra's algorithm, 27
- Fibonacci number, 80
- finishing time, 87, 179
- first-in, first-out, 25, *see also* queue, 57
- flow network, 199
- fractional knapsack problem, 182
- free tree, 87, 87
- full binary tree, 60, 60
- function
  - hash, *see* hash function
  - probability density, 100
  - recursive, 29



- garbage collection, 95
- Gaussian random variable, 100, 101
- GCD, 47, 47, 115, 116
- generalization principle, 30
- graph, *see also* undirected graph, 40, 44, 78, 85,  
*see also* directed graph, 85, 86, 87,  
*see also* tree, *see also* flow network
  - bipartite, 89
- graph modeling, 36, 40, 87
- graphical user interfaces, *see* GUI
- greatest common divisor, *see* GCD
- greedy, 19, 28, 31, 31, 33, 35, 78, 83
- GUI, 95
  
- Hamiltonian cycle, 91
- Hamiltonian path, 105
- hash code, 33, 34, 71, 71, 150, 189
- hash function, 23, 26, 26, 33, 34, 71, 143, 150
- hash table, 3, 20, 22, 23, 23, 26, 27, 30, 51, 71, 72,  
123, 143, 149, 150, 168, 177, 187, 193
  - deletion from, 23, 71
- head
  - of a deque, 57
  - of a linked list, 53, 54, 122–125
  - of a postings list, 55, 127
  - of a queue, 57, 131–133
- heap, 22, 23, 23, 26, 27, 63, 63, 73, 80, 139
  - Fibonacci heap, 27
  - max-heap, 63, 73
  - min-heap, 63, 73
  - priority queue, 26
  - treap, 27
- heapsort, 73
- height
  - of a binary search tree, 76, 77, 162
  - of a binary tree, 23, 25, 26, 60, 60, 61
  - of a complete binary tree, 60
  - of a domino, 42
  - of an event rectangle, 74, 75
  - of a line segment, 27
  - of a perfect binary tree, 60
- height-balanced, 27
- height-balanced tree, 27
- HTTP, 70, 100
- Huffman code, 84, 172–175
- Huffman tree, 173, 174
- Hypertext Transfer Protocol, *see* HTTP
  
- I/O, 19
- IDE, 13
- in-place sort, 73
- incremental improvement, 28, 31, 32
- indirect sort, 152
- inequality
  - linear, 91
- integral development environment, *see* IDE
- Internet Protocol, *see* IP
- interval tree, 23
- intractability, 4, 91
- invariant, 36, 41, 43
- inverted index, 74
- IP, 70, 70, 147
- iterative refinement, 36, 37, 38
  
- knapsack problem
  - 0-1, 93, 181
  - fractional, 182
- Kruskal's algorithm, 27
  
- LAN, 79
- last-in, first-out, 25, *see also* stack, 56
- LCA, 62, 62, 135
- leaf, 23, 60, 61, 172–175
- left child, 57, 59, 60, 87, 134, 135, 159, 161
- left subtree, 26, 59–61, 76, 159, 161, 162
- Levenshtein distance, 36, 82, 82, 96, 166, 167, 168,  
185
- line segment, 27
  - height of, 27
- linear inequality, 91
- linear programming, 43, 199
  - simplex algorithm for, 43
- linked list, 23, 25, 201
- list, 23, *see also* singly linked list, 54–57, 71, 122,  
123, 125–127, 139
  - postings, 55, 127
- livelock, 96
- load
  - of a hash table, 71
- local area network, *see* LAN
- lock
  - deadlock, 96
  - livelock, 96
- longest alternating subsequence, 166
- longest bitonic subsequence, 166
- longest convex subsequence, 166
- longest nondecreasing subsequence, 82, 82, 164,  
164
- longest path, 79, 163, 174, 178
- longest weakly alternating subsequence, 166
- lowest common ancestor, *see* LCA
  
- matching, 89
  - bipartite, 199
  - maximum weighted, 89
- matrix, 87, 98
  - adjacency, 87
  - multiplication of, 95, 189
- matrix multiplication, 95, 189
- max-heap, 63, 73, 137, 138

- deletion from, 23, 63
- maximum flow, 89, 89
- maximum weighted matching, 89
- mean, 101
- median, 34, 38
- merge sort, 73, 78
- min-heap, 23, 26, 33, 63, 73, 136, 139, 140, 187
  - in Huffman's algorithm, 172
- minimum spanning tree, 27, *see* MST, 89, 89
  - Kruskal's algorithm for, 27
- Morris traversal, 61, 61
- MST, 78, 79
- multicore, 95
- mutex, 96
- mutilated chessboard, 29
- network, 95
  - local are network, 79
  - network bandwidth, 33, 70
  - network route, 64
  - network session, 101
- network bandwidth, 33, 70
- network session, 101
- node, 79, 163
- nondecreasing subsequence, 165
- NP, 91
- NP-complete, 35, 93, 181
- NP-hard, 83
- null string, 174
- open interval, 156
- operating system, *see* OS
- order statistics
  - dynamic, 23
- ordered pair, 150
- ordered tree, 87, 87
- OS, 4, 98
- overflow
  - integer, 40, 67
- overlapping intervals, 156
- palindrome, 168
- parallel algorithm, 92
- parallelism, 28, 33, 95, 96
- parent-child relationship, 60, 87
- partition, 33, 72, 78, 148, 169, 188, 189
- path, 85
  - shortest, *see* shortest paths
- PDF, 9
- perfect binary tree, 60, 60
  - height of, 60
- permutation, 102, 170, 190, 193
  - random, 101, 102
- Poisson random variable, 100
- Portable Document Format, *see* PDF
- postings list, 55, 55, 127
- power set, 46, 46, 47, 113
- prefix
  - of a string, 84, 174, 175
- prefix sum, 39
- primality, *see* prime
- prime, 43, 76
- priority queue, 26, 26
  - deletion from, 26
- probability density function, 100, 100
- production sequence, 88, 177
- queue, 23, 25, 26, 57, 57, 58, 131–133, 161, 176, 177
  - priority, 26
- quicksort, 3, 24, 44, 49, 73, 78, 81, 100
- race, 96, 185
- radix sort, 73
- RAM, 33, 63, 64, 70, 136, 137, 147, 189
- random access memory, *see* RAM
- random number generator, 101, 190, 191
- random permutation, 101, 102
- random variable, 100, 100
  - Bernoulli, 100
  - Gaussian, 100, 101
  - Poisson, 100
  - uniform, 100
- randomization, 28, 28, 33, 34, 71, 78
- randomized algorithm, 43
- reachable, 85, 87
- recursion, 12, 28, 29, 29, 30, 31, 41, 53, 57, 61, 80, 116, 121, 126, 131, 170, 182
- recursive function, 29, 29
- red-black tree, 26, 27, 76
- reduction, 36, 39, 78
- regular expression, 29, 51, 51, 122, 168
- rehashing, 71
- Reverse Polish notation, 25
- right child, 57, 59, 60, 87, 134, 135, 159, 161
- right subtree, 26, 59–61, 76, 134, 159, 162
- root, 57, 59–62, 76, 87, 131, 133–135, 137, 159, 161, 162, 172–174
- rooted tree, 87, 87
- scheduling, 89, 179
- searching
  - binary search, *see* binary search
- sequence, 165–167
  - alternating, 165
  - ascending, 166
  - bitonic, 166
  - convex, 166
  - production, 88, 177
  - weakly alternating, 166

- shared memory, 95, 95
- Short Message Service, *see* SMS
- shortest path, 88–91, 163, 176, 179, 181
  - Dijkstra’s algorithm for, 3, 27, 179
- shortest path, unweighted case, 176
- shortest paths, 88
- shortest paths, unweighted edges, 177
- sibling, 175
- signature, 34
- simplex algorithm, 43
- singly linked list, 23, 25, 53, 53, 54
- sinks, 86
- skip list, 27
- small example, 36, 37, 194
- SMS, 96
- social network, 13
- sorting, 28, 29, 33, 34, 38, 43, 49, 64, 68, 70, 73, 78,  
136, 137, 152, 155, 156
  - counting sort, 50
  - heapsort, 73
  - in-place, 73
  - in-place sort, 73
  - indirect sort, 152
  - merge sort, 73, 78
  - quicksort, 24, 44, 49, 73, 78, 81, 100
  - radix sort, 73
  - stable, 73
  - stable sort, 73
- sources, 86
- space complexity, 2
- space constraint, 35
- spanning tree, 87, *see also* minimum spanning tree
- SQL, 16
- square root, 32, 43, 69
- stable sort, 73
- stack, 23, 25, 56, 56, 61, 126, 128–131, 176
- Standard Template Library, *see* STL
- starvation, 96
- state, 28
- STL, 76
- streaming
  - algorithm, 44
- string, 23, 23, 26, 27, 29, 36, 38, 39, 47, 51, 52, 72,  
82, 84, 88, 94, 96, 114, 120, 121, 149,  
166–169, 172, 177, 183, 185, 187
  - null, 174
- string matching, 29, 51, 82
  - Boyer-Moore algorithm for, 39
- strongly connected directed graph, 86
- Structured Query Language, *see* SQL
- subarray, 2, 34, 39, 49, 80, 81, 117, 118, 137, 141,  
162, 190
- subsequence, 27, 164, 165, 167
  - longest alternating, 166
  - longest bitonic, 166
  - longest convex, 166
  - longest nondecreasing, 82, 164
  - longest weakly alternating, 166
  - nondecreasing, 165
- subset sum, 181
- substring, 52, 121, 168
- subtree, 60, 159, 161–163
  - left, *see* left subtree
  - right, *see* right subtree
- Sudoku, 93, 94, 183
- suffix, 51
- synchronization constraint, 186
- tail
  - of a deque, 57
  - of a linked list, 53, 123, 125
  - of a queue, 57, 132, 133
- tail recursion, 79
- tail recursive, 80, 127
- time complexity, 2, 11
- timestamp, 26, 64
- topological order, 179
- topological ordering, 86, 178, 179
- total order, 64
- tour, 32
- treap, 27, 27
- tree, 87, 87
  - AVL, 26, 27
  - BFS, 176
  - binary, *see* binary tree
  - binary search, *see* binary search tree
  - diameter, 79, 80, 163
  - free, 87
  - Huffman, 173, 174
  - interval, 23
  - ordered, 87
  - red-black, 26, 27, 76
  - rooted, 87
  - treap, 27
- trie, 27, 27
- triomino, 29, 30
- UI, 19, 95
- undirected graph, 27, 41, 86, 86, 87, 89, 175, 177
  - weighted, 78
- uniform random variable, 100
- Uniform Resource Locators, *see* URL
- URL, 9, 82, 189
- user interface, *see* UI
- variance, 100, 101
- variation, 36, 41
- vertex, 40, 41, 78, 85, 85, 86–90, 175–179, 189, 199,  
201

connected, [86](#)

weakly alternating sequence, [166](#)

weakly connected graph, [86](#)

weighted undirected graph, [78](#), [79](#)

write an equation, [36](#), [40](#)